Continuous development and release automation of web applications

Miikka Eloranta

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 15.01.2018

Thesis supervisor:

D.Sc. Valeriy Vyatkin

Thesis advisor:

M.Sc. Juha Männikkö

Aalto University
School of Electrical Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Miikka Eloranta

Title: Continuous development and release automation of web applications

Date: 15.01.2018 Language: English Number of pages: 7+63

School of Electrical Engineering

Professorship: Control, Robotics and Autonomous Systems

Supervisor: D.Sc. Valeriy Vyatkin

Advisor: M.Sc. Juha Männikkö

Software development methods have evolved towards more agile practices for getting changes implemented more quickly. While the new features are finished faster, the release processes are also made more automated to get the changes in production environment reliably, repeatably and rapid.

This thesis examines the practices used for continuous software development and web application release automation. The main objective is to find out and implement a way for making changes agilely and getting them tested and released in several environments effortlessly.

After the research, different tools are sought for, compared and suitable tools are selected. Lean software development is chosen as the working practice for the development. GitHub enterprise is used for version control, JetBrains TeamCity for continuous integration and Octopus Deploy for deployment automation. SonarQube is used for static code analysis and UseTrace for automated functionality testing.

The lean development practice is found well fit for real world use. The deployment pipeline is also well operational, founding issues early and deployments are enabled steady, effortless and fast. Some issues with code analysis are found due to the decisions in the application implementation. UseTrace tests have occasionally some false positives in the failing test results but overall they worked as expected.

Keywords: continuous software development, deployment pipeline, agile, continuous integration, continuous delivery

AALTO-YLIOPISTO
SÄHKÖTEKNIIKAN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Miikka Eloranta

Työn nimi: Verkkosovellusten jatkuva kehitys ja julkaisuautomaatio

Päivämäärä: 15.01.2018 Kieli: Englanti Sivumäärä: 7+63

Sähkötekniikan korkeakoulu

Professuuri: Säätötekniikka, robotiikka ja autonomiset järjestelmät

Työn valvoja: D.Sc. Valeriy Vyatkin

Työn ohjaaja: M.Sc. Juha Männikkö

Sovelluskehitysmenetelmät ovat kehittyneet ketterämmiksi, jotta muutokset saataisiin toteutettua nopeammin. Kun uudet ominaisuudet valmistuvat nopeammin, myös julkaisuprosesseista tehdään automatisoidumpia, jotta muutokset saadaan tuotantoon luotettavasti, toistettavasti ja nopeasti.

Tässä työssä tutkitaan menetelmiä, joita käytetään jatkuvaan sovelluskehitykseen ja verkkosovellusten julkaisuautomaatioon. Pääasiallisena tavoitteena on selvittää ja toteuttaa tapa, jolla muutoksia voidaan tehdä ketterästi ja julkaista testattuina moniin ympäristöihin pienellä vaivalla.

Tutkimuksen jälkeen etsitään erilaisia työkaluja, joita vertaillaan keskenään ja soveltuvat työkalut valitaan. Lean-malli valitaan sovelluskehityksessä käytettäväksi tavaksi. GitHub enterprise:a käytetään versionhallintaan, JetBrains TeamCity:ä jatkuvaan integraatioon ja Octopus Deploy:ta jatkuvaan toimittamiseen. SonarQube:a käytetään staattiseen koodianalyysiin ja UseTrace:a automaattiseen funktionaalisuustestaamiseen.

Lean-sovelluskehitysmalli todetaan hyvin toimivaksi todellisessa käytössä. Julkaisuputki on myös hyvin toimiva, löytäen ongelmat ajoissa ja mahdollistaen julkaisut luotettavasti, vaivattomasti ja nopeasti. Koodianalyysin osalta joitain ongelmia ilmenee sovellustoteutukseen liittyvistä päätöksistä johtuen. UseTrace-automaattitestit tuottavat satunnaisesti virheellisiä ongelmia testituloksissa, mutta yleisesti ottaen ne toimivat odotetusti.

Avainsanat: jatkuva kehitys, julkaisuputki, ketterä, jatkuva integraatio, jatkuva toimitus

Preface

First of all, I would like to thank both of my supervisors, D.Sc. Ilkka Seilonen and D.Sc. Valeriy Vyatkin from Aalto University, who gave me valuable feedback during this prolonged thesis process.

I would also like to thank the whole development team working on the application and the continuous deployment pipeline implemented, for following the agreed processes and making suggestions for improvements, as well as supporting me with the thesis.

Finally, I would like to thank all my friends and family for the support they provided and for withstanding me babbling about the difficulties while having a beer or two with me.

Otaniemi,

Miikka Eloranta

Contents

Abbreviations

ALM	Application Lifecycle Management
API	Application Programming Interface
ARA	Application Release Automation
ASR	Architecturally Significant Requirement
ASD	Adaptive Software Development
BT	BizTalk
Cake	C# Make
CAS	Complex Adaptive System
CSS	Cascading Style Sheets
CD	Continuous Delivery
CI	Continuous Integration
DoD	Definition of Done
DevOps	Development and Operations
DSL	Domain-Specific Language
E2E	End-to-End
HTML	HyperText Markup Language
IaC	Infrastructure as Code
IIS	Internet Information Service
IP	Internet Protocol
IT	Information Technology
JS	JavaScript
JSON	JavaScript Object Notation
LB	Load Balancer
LSD	Lean Software Development
LXC	LinuX Container
MVP	Minimum Viable Product
NoOps	No Operations
NPM	Node.js Package Manager
OOP	Object-Oriented Programming
OS	Operating System
QA	Quality Assurance
RDP	Remote Desktop Protocol
SaaS	Software as a Service
SCM	Software Configuration Management
SLA	Service Level Agreement
SSH	Secure SHell
SSL	Secure Sockets Layer
SFTP	SSH File Transfer Protocol
TDD	Test Driven Development
TS	TypeScript
UAT	User Acceptance Testing
UI	User Interface
VCS	Version Control System
WHATWG	Web Hypertext Application Technology Working Group
W3C	World Wide Web Consortium
XML	eXtensible Markup Language
XP	eXtreme Programming

1 Introduction

1.1 Background

At the beginning of the millennium, Highsmith and Cockburn noted that in many software development processes developers were more concerned about the customer satisfaction at the time of delivery than conforming a successful plan [1]. For the time being, this might seem like a good idea but the "technical debt"[1] this causes is more likely to cause more trouble than momentary customer satisfaction is worth. Even as early as 1993 so called Barry Boehm's life cycle cost differentials theory was coined, saying that change grows through software's development lifecycle [3]. While inevitable fast changes are often required, the solution is not to stop change but the life cycle cost differentials theory must be taken into account, so these changes would not cause problems in the future [1].

For applications and systems, there are often requirements for updates and changes. There might be some newly found vulnerability that needs to be fixed or some new features are wanted. In many cases, this raises the issue of rebuilding and installing the software and, especially on web applications, this might require a maintenance break. Even though this process can be quick nowadays, it must be ensured that everything happens without issues, which causes the need for someone to watch over the process. If the web application has a lot of usage, it might be necessary to manage the maintenance break at a quiet time of usage (e.g. in the middle of the night).

Efficiency of software development has recently increased, due to plenty of different reasons. The methods such as extreme programming, lean development, adaptive software development, continuous integration and test driven development are leading developers to more dynamic, rapid and effective workflows, where the issues are detected and identified at an early state of development. Software development is nowadays a continuous process and the goal is no more to eventually stop the change but to keep the software working and stable through its lifetime with good development, building and testing practices. The collaboration between software developers and other information technology (IT) operators has also grown, while automation and testing tools have also evolved. While virtualization and cloud computing are also replacing physical servers, the co-operation of programmers and infrastructure operators achieves rapidly changeable dynamic infrastructures. Infrastructures can also be managed like software and given the automation possibilities this provides, it reduces the need for long maintenance breaks and human involvement.

1.2 Objectives

The objective of this thesis is to develop a continuous application release process. That contains practical use of version control, quality analysis, automated application

[1]The metaphor "technical debt" was first coined by Ward Cunningham in 1993 to describe the situation in which a short-term gain in development speed is gotten at the expense of long-term code quality [2].

building and testing the components, both as separate units and as a complex, as well as the actual release of the new version of the application. The target is set for the release process to be a rapid, repeatable and reliable process during the release over old version. This thesis concentrates on web-based applications that are connected to via web browser. Although, web-based applications are the main focus for this thesis, many of these practices can be – and already are – used with other types of applications as well.

For discovering an appropriate process, it must be found out, what types of methods, applications and platforms can be used to reach this goal. Different approaches must also be well and unbiasedly compared. For each part of the process the most fitting method must be found. The variety of terminology used in software development – as well as its structure – must also be understood clearly. Although, due to a large variety of different procedures and practices in software development, not all of them will be included in this thesis but some – carefully selected – will be better presented, as well as the larger idea behind continuous software development practices used nowadays. For the sake of the importance of collaboration between software developers and infrastructure operators, the methods for infrastructure operations' side must also be discussed, although, it will not be included in the release process to be implemented.

The focus of this thesis is how to implement a software development and deployment process that provides developers with agile methods, compiles the source codes, automatically runs necessary tests, builds and deploys it to necessary manual testing environments and finally releases it to production environment over the old version. The process should not require developers to connect to any specific environment or server but just to the user interface used for the continuous integration and deployment tools. While this thesis concentrates on web-based applications for clients' businesses, the release over old version should not cause long outages for the application, so the application would be reachable soon after the release process is started. Even though, the process for the release is going to be automated, the actual release to production environment will be started manually. The application owner must be able to decide, when the release is done, even though the working tested version would be ready earlier. Mostly the customer would also want to perform user acceptance testing and quality assurance, where all the developed new features are tested in a test environment by a human user to ensure everything is working as expected, before the actual release.

1.3 Methods

For finding out the best methods for continuous software development as well as implementing an automated application release process, a literature survey is required. In this survey, alternative methods and practices are examined and existing critique, comparisons and studies are surveyed. Possible standards for separate parts of the process are also looked for. Already existing processes will also be surveyed and compared and the possible improvement for those will be studied.

For the actual implementation part, the practices and tools chosen based on this

study are used. The compatibilities of these are also reviewed. The difficulties and realizations about possible incompatible decisions will be documented and better alternatives put in use instead.

The final implementation is also tested by using the chosen practices and methods, as well as using the implemented continuous delivery pipeline to create new release. The success in according to the expectation is discussed.

1.4 Thesis structure

The second chapter of this thesis introduces programming practices such as extreme programming and test driven development that are used in continuous software development nowadays. First, it generally discusses continuous software development, then describes some components of it and finally some chosen methods for the development process. It describes the origin of those methods and compares them with one another. The benefits and weaknesses of the each are also explained.

The third chapter describes the methods for releasing applications automatically. The overall process and components for releasing are introduced, including the whole pipeline from the change in source code to releasing it to production environment. Risks throughout the process are also explained.

The fourth chapter lists and compares different tools for implementing the whole pipeline – step by step – starting from the change made in code and ending with the release in production.

In the fifth chapter, a practice for continuous development and a tool or platform for each step of the process is chosen, and the choice is reasoned. It is also ensured that there are no incompatibilities between the tools chosen. The chapter describes the actual implementation of the whole system, given the tools and platforms chosen in the fourth chapter. The tools are initialized and possible conflicts or problems during the initialization are discussed. The system components are configured and set up and finally the overall process for a code change is tested with a web application programmed with .NET framework with additional front-end development libraries.

The sixth chapter describes the actual usage and testing of the pipeline implemented in the fifth chapter. By actually using the tools or platforms provided together with the software development practices chosen, it concludes the usage of them and describes how it went. It also suggests some improvements for it, given the practical experience of using them.

The seventh chapter summarizes what has been done and makes conclusions about how the objectives of the thesis were achieved. Differing choices about tools are also discussed. Finally, the possibilities for further development and improvements are considered.

2 Continuous programming practices

In traditional software development, the objective is to plan the software carefully, develop it and fix the defects, so the software could be completed and the change stopped [1]. The software has been developed using so called waterfall model [4]. In that model, users would anticipate the complete set of requirements for the software, then it would be designed, developed and tested as a whole, as shown in Figure 1 [1, 4]. It is assumed that the specifications are good enough and the complete set of requirements can be anticipated early, and based on these the software can be developed. This approach assumes that all variation from the original plan is result of errors.

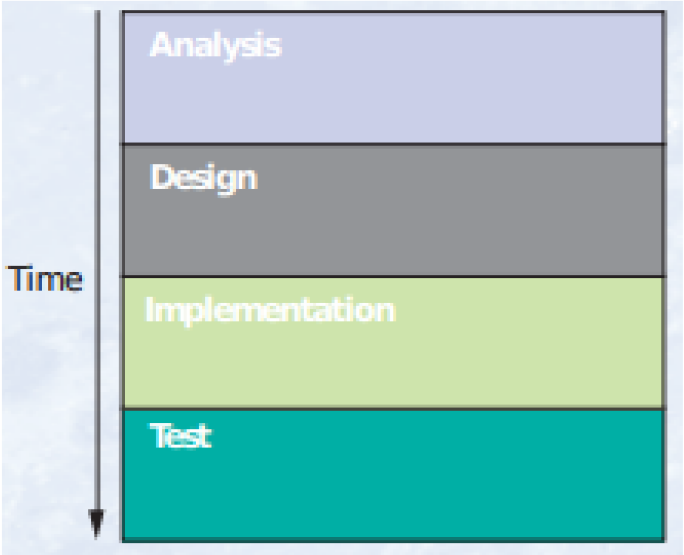


Figure 1: Traditional waterfall model in software development [4].

The problem with this model is that the users do not tell all they want at the beginning [4]. They either do not know exactly what they want, or they could contradict themselves. Another problem with this model is that developers might not be able to successfully plan very complex applications entirely. With large software, this might lead to a realisation after half the planned time has been used that only a small fraction of it is actually completed.

The issues with too large development cycles in the waterfall model lead to iterations [4]. Figure 2 shows how the development cycle was cut into smaller pieces. This way the complexity of the software designed and developed at once is reduced and fulfilment of the plan is more likely to succeed. The academic software engineering community was still not convinced in this development [4]. The high cost of changing software was taken as a challenge and some more cost reduction model than just cutting the waterfall into smaller pieces was tried to be achieved [4]. This lead to continuous development practices – especially extreme programming (XP) at the time – which make the changing nature of software an assumption instead of thinking it as a result of problems [1, 4].

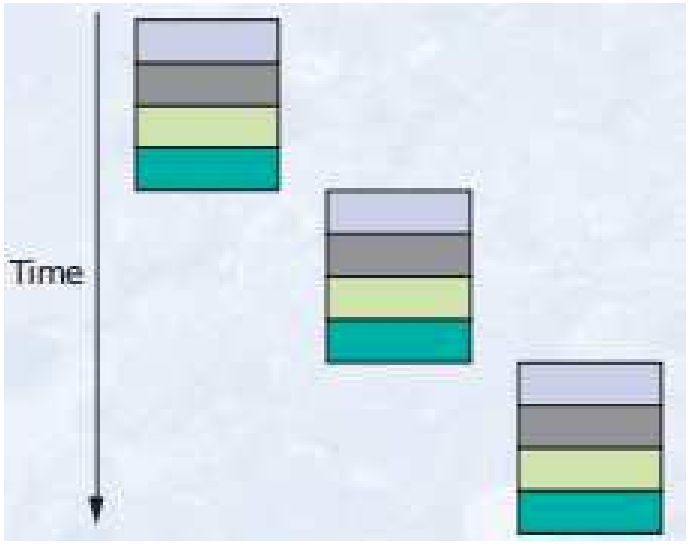


Figure 2: Waterfall iterations for shorter development cycles [4].

In continuous software development practices, changing the software is a continuous process done a small part at a time [1]. While the business conditions are continuously changing, businesses must adapt to new conditions also in software development. For all continuous software development methods the main hallmarks are team proximity and intense interaction between team members [1]. Highsmith and Cockburn pointed out that eliminating change early is leaving the business unresponsive for business conditions which leads to business failure [1].

The continuous change in software development can certainly cause process problems that lead to errors but external changes also exists e.g. in the environments that lead to critical variation. These changes cannot be completely eliminated and the new strategy is to reduce the cost of responding to these variations and possible errors by making the development a continuous process made a small change at a time [1]. The main difference between traditional software development methods and continuous – also known as agile – software development methods is that continuous methods places more value on the planning process than the documentation it results.

Even though, the use of agile methods has increased a lot recently, it is not to say that all traditional – also called plan-driven [5] – software development is a worse alternative. Continuous methods rely on the tacit knowledge embodied in the team rather than writing the knowledge down in well documented plans, which may cause problems especially with unexperienced junior developers [6]. As Barry Boehm points out, a significant consideration is the unavoidable statistic that near half of the world's software developers are below average [5]. Highsmith and Cockburn also list the critical people factors for successful agile development as: amicability, talent, skill, and communication [7]. As for managing the developers while using agile methods, Larry Constantine points out another problem: "There are only so many Kent Becks in the world to lead the team. All of the agile methods put a premium on having premium people" [8]. As all of these notes point out, ultimately the most important factors in successful software development are the people developing the

software.

2.1 Version control

Version control, also known as revision control, source control or software configuration management (SCM), is a practice in which developers have a centralized repository containing the definite version of files and history of changes in these files [9]. Systems for version control are called version control systems (VCS) and they are essential for software development – especially for multi-developer projects [9]. With version control system each developer *pulls* – or initially *checks out* or *clones* – the latest source code from the repository to their private environment – their own computer – and starts developing on that [10]. After a bug is fixed or a new feature developed, a change – called a *commit* in VCS terminology – is made and *pushed* back to the central repository [10]. When multiple developers are modifying the same file, VCS *merges* only the changed lines or warns about the conflict that the file has changed from the original version pulled [10].

A good development practice is to maintain all the source code, build scripts, translations, etc., in a VCS [10]. Each change is put in separate commits to find the modifications or additions made later on [10]. File naming should initially be carefully thought of while many systems manage renaming as removal and addition of a file, making it a bit harder to seek all the changes made to such file in the future [10]. All commits included in a single release should also be *tagged* or *labeled* clearly [10]. Separate features, bug fixes etc. can also be put in different *branches* to separate features and avoid multiple conflicts during development [10]. Different branches can be merged into the default branch – *trunk* – afterwards [10].

Although, it might be a good idea to use different branches for different features, separate branches should not live their separate lives for too long [11]. If each developer uses their own branch or branches and makes many commits with big changes to them for long, the possibility of merge conflict increases [11]. If the same file has been changed in multiple branches, this will probably cause merge to either fail or some developers' changes will revert the changes of someone else. The work buildup caused by using feature branching with too long living branches is shown in Figure 3 [11]. Morris, as well as well as Humble and Farley, also recommend not using separate branches in software development but the reasons are explained later on chapter 3.2 [11, 12].

2.2 Code review

While in agile methods the whole team is well included in the development and everyone on the team should be able to continue on any part of the software, a peer code review is mostly used when committing new changes [13]. In code review, a new change in source code is reviewed by someone else than the person that has written the change [13]. Code review is intended for finding mistakes overlooked in initial code development phase and for increasing the software quality.

Code review practices are divided to two main categories: formal code review and lightweight code review [13]. In formal code review, all the developers attend a series of meetings and review code line by line – and mostly the source code is printed out to papers [13]. In formal code review the source code is inspected extremely thoroughly and it is proven effective in finding defects in the code [13].

Lightweight code review is more used in agile methods due to a shorter duration of time taken from the actual application development. Lightweight reviews are concluded as a part of the normal development process and different types of these reviews are [13]:

- **Over-the-shoulder**: The developer walks through the code he created as another person – "the reviewer" – follows through and listens.
- **Email pass-around**: The VCS sends an email with a changed source code to reviewers after a commit has been made.
- **Tool-assisted code review**: An external tool is used for reviewing new changes in source code. This does not specify, what type of tool is used but specialized tools designed for code review are mostly used nowadays.
- **Pair programming**: Two developers are developing the code simultaneously on the same computer, so the other one not typing automatically reviews the changes.

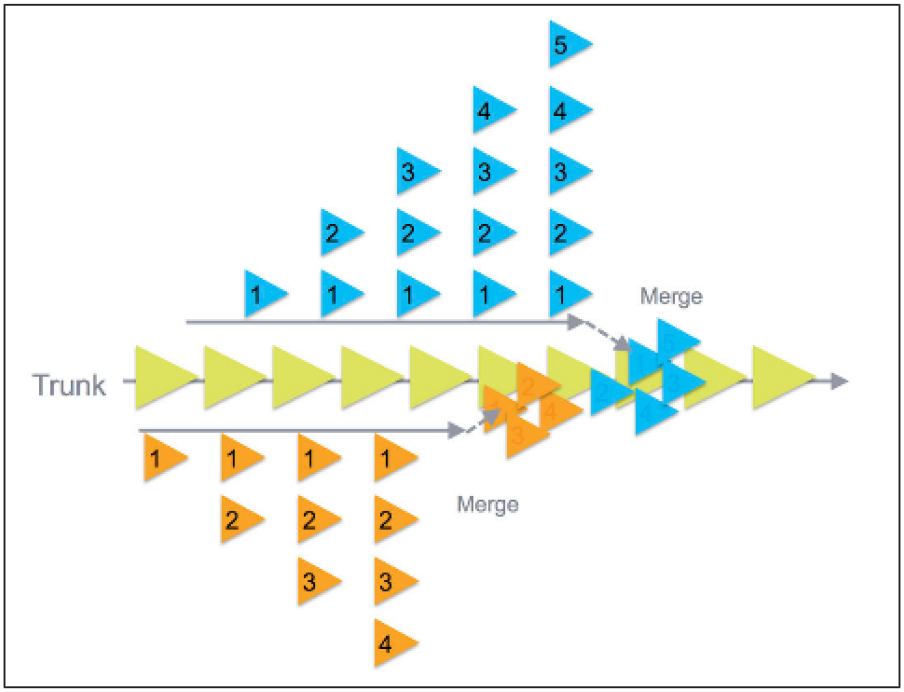


Figure 3: Too long lifetime of separate feature branches builds up work and causes issues when merging [11].

From these lightweight code review types, it would be fair to assume that pair programming reduces the speed of development by 50% while two developers are using the same computer. Williams et al. showed that pair programmers can complete a task 40% faster than a single programmer and develop better code and algorithms [14]. At the beginning of the study by Williams et al., pair programmers took 60% more time for programming but after the adjusting period – losing the urge to grab the mouse and keyboard from their partner – the time reduced to a minimum of 15% and by working in tandem the pairs finished their tasks 40% to 50% faster [14]. Also, it is quite clear that when two people are looking at the same code, the misspellings and logical errors made are more easily noticed.

A slightly similar idea is behind so called "rubber duck debugging," where the attempted functionality of inoperative code is thorougly explained, line by line, to anyone or anything (e.g. a rubber duck on your desktop), making the developer think about the code more thoroughly [15]. This is not a type of review itself, though. It is just a method for finding the cause of problems. However, when using over-the-shoulder type of review, it is, by definition, a form of rubber duck debugging.

2.3 Test driven development

In test driven development (TDD) the developer must first think of the test cases that the software must succeed. The unit tests are the first things to be implemented and not until they are done will the actual functionality development to manage those tests begin [16]. This does not mean that all the tests would be written before programming the functionality, though. The main idea in the TDD is that the developer must think the behaviour of the software in different situations beforehand and then write the tests and functionality for it part by part [16]. TDD has already been sporadically used in software development for decades [17].

Robert C. Martin coined the three laws of TDD as [18]:

1. You may not write production code until you have written a failing unit test.
2. You may not write more of a unit test than is sufficient to fail, and not compiling is failing.
3. You may not write more production code than is sufficient to pass the currently failing test.

These three laws sum up the TDD pretty well. Unit tests must be written first, they should not be too complex and the test and functionality are written in small parts – single tests and their functionalities at a time.

TDD does not limit to only unit tests [16]. The communication between components must also be taken into account [16]. When developing e.g. a new service that is used by either new or already existing component, the unit tests are developed in the beginning. After the unit tests are completed, the functional tests are developed to ensure that the component or components interacts with the service successfully [16]. Not until then, the new service and possibly new component are developed.

2.4 Extreme programming

In extreme programming (XP), the traditional waterfall model – or its iterations – seen in Figures 1 and 2 are turned sideways and cut into even smaller pieces [4]. The software is simultaneously analysed, designed, implemented and tested a small piece at the time [4]. The XP model is shown in Figure 4. In the XP model the customer picks the most relevant features – *stories* for programmers – for next iteration [4]. The stories are then cut into *tasks* by developers and each task is given an estimated development duration [4].

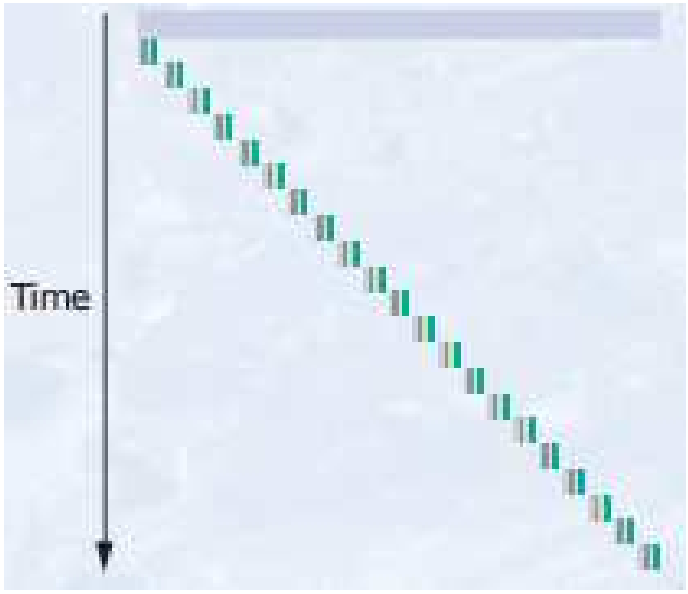


Figure 4: Waterfall model process turned sideways, called extreme programming [4].

Beck lists the 12 core practices of extreme programming as [19]:

- **Planning game**: Customers decide the scope and timing of releases based on programmers' estimates. Programmers implement only the functionality demanded.
- **Small releases:** New releases are made often and the pipeline from planning to production is accomplishable in a few months.
- **Metaphors**: The shape of the system is defined by a metaphor or metaphors shared between programmers and customers.
- **Simple design**: Only the fewest possible classes and methods – no duplicate methods. Design must run all tests at every moment.
- **Testing**: Unit tests are written all the time. All tests must run correctly. Customers define functional tests that should also all run.
- **Refactoring**: The design is evolved through transformations so that all the tests in existing design keep running, even though a change to its functionality is required.

- **Pair programming**: All code is programmed by two people on a single computer. Pair programming is more thoroughly explained in chapter 2.2.
- **Collective ownership**: Any programmer can change any line of the code. No bottlenecks.
- **Continuous integration**: New code is integrated to current system after no more than a few hours. When integrating, the whole system is built from scratch and all the tests must pass or the change is discarded.
- **40-hour weeks**: Nobody works overtime. Even rare overtime occurrences are a sign of deeper problems.
- **On-site customers**: Customer sits with the developers full time.
- **Coding standards**: Code is always written formatted by agreed standards.

.

In addition to these core practices, test driven development (TDD) has recently been raising added visibility as a practice of XP [19, 16]. TDD process is explained in chapter 2.3.

2.5 Lean software development

The term "lean software development" (LSD) – often referred to simply as "lean" – originates in a book by the same name, written by Mary and Tom Poppendieck [20]. LSD is a translation from lean manufacturing and lean IT to software development. The philosophy of lean manufacturing derived from the Toyota Production System, where the concept is to make only what is needed, only when it is needed and only in the amount that is needed using automation with a human touch [21]. As in lean manufacturing, in LSD the developers try systematically to get rid of waste caused by e.g. uneven workloads and unnecessary work [20].

The Poppendiecks summarize LSD to seven principles, that are very close in concept to lean manufacturing process [20]:

- **Eliminate waste**: Everything that does not add value to the customer is regarded as waste and should not be done. This includes extra features and processes, bugs implemented due to carelessness, work only partially done, switching between tasks too often and unnecessary management.
- **Amplify learning**: Software development is both writing the code and learning about the problems been solved in short iterations. Different ideas can be tried and the value of software is measured by the fitness for use and not strictly following the requirements.
- **Decide as late as possible**: While in software development it is uncertain which the best approach is, different options are reviewed and tested for as long as possible. This way the alternatives gain more facts behind them until the final decision is done.

- **Deliver as fast as possible**: The shorter the iteration periods are, the more the developers learn from them and customers' current needs are fulfilled instead of such made a while ago or are planned for the future.
- **Empower the team**: The traditional decision-making is turned upside-down and the managers listen to the developers ideas for how to make team work better. The aphorism "find good people and let them do their own job" is favored in lean [22].
- **Build integrity in**: While the iteration cycles are kept small, the problems and implementations of the iteration can be better explained to the customer at the same time the problem is solved or a new feature developed. Refactoring is encouraged for keeping the architecture integral.
- **See the whole**: The relationships between different components must be well defined. The whole structure must be well known by all the developers to be able to implement fast. The Poppendiecks framed the slogan "think big, act small, fail fast; learn rapidly" that summarizes the importance of understanding the field and the suitability of implementing lean principles.

So, in lean development, the definition of done (DoD) is set for the minimum viable product (MVP) where only the most necessary features are developed. After that is finished and DoD fulfilled, the next version of an MVP is defined by some new features for further development. The requirements are filled by the programming practices chosen by the team to be the best fit.

2.6 Adaptive software development

A slightly differently structured traditional waterfall model than the one seen in Figure 1 is shown in Figure 5. The waterfall model is characterized by linearity and predictability with almost no feedback about the process at all. At the end of the last millennium, this model was the most used in software development [23]. In the mid-1980 a spiral model by Barry Boehm as well as an evolutionary model by Tom Gilb emerged [23, 24]. Evolutionary and spiral models' life cycle is shown in Figure 6. As it can be clearly seen, this model offers better feedback and predictability than the traditional waterfall model.

Figure 5: The life cycle of traditional waterfall model [23].

With adaptive software development (ASD) model, the lifecycle is built on a different world view [23]. Although, it has the same cyclical form, it reflects the unpredictable domain of increasingly complex software, as can also be seen from the

phase names in Figure 7. Where spiral and evolutionary models assume determinism, ASD replaces it with emergence. It also takes into account a deeper level of change than just those in its life cycle [23]. When spiral or evolutionary model would look for a cause-to-effect paired rule from a largely changed environment, adaptive model knows that such rule does not exist.

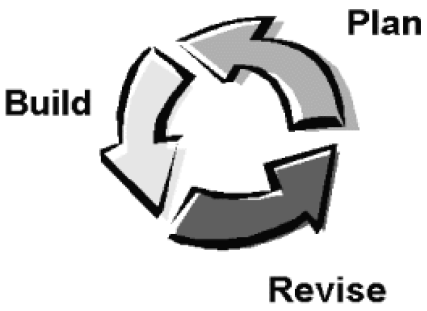


Figure 6: The life cycle of spiral and evolutionary model [23].

The steps in ASD life cycle are built on uncertainty [23]. In complex environments, planning is a paradox, as based on complex adaptive systems (CAS) theory, outcomes are unpredictable [25]. That is why the *plan* phase from former models is replaced with *speculate* in ASD. Also, when it is not possible to predict and plan the outcome, it is also not possible to have total control over the system. That is why the next phase is called *collaborate*. As *learning* has a simple definition in the dictionary as "the activity or process of gaining knowledge or skill by studying, practicing, being taught, or experiencing something" [26], adaptive model challenges both developers and customers to examine their assumptions and use the results for learning. It can be seen that the phases of ASD life cycle do overlap. It is difficult to collaborate without learning or learn something without collaborating. They are on purpose nonlinear and overlapping because that is the only good way to define adaptivity – at least based on the ASD model.

Figure 7: The life cycle of adaptive software development model [23].

2.7 Comparison of continuous software development methods

Whereas the XP and the LSD are actual iterative continuous software development methods, ASD is more of an approach to software development. The ASD gives developers the idea of the nature of software development nowadays, where it is not addressed to as some application with a finalized description from the beginning. With ASD it is always unknown, to which way the software should develop to, and to what end would it be "finalized". There is too much variety for the needs of users to get some specific final specifications for the software in the end.

The XP and the LSD methods use this way of thinking as the base of their practices. Both of these practices use iterative methods with short cycle times and are well adapted for continuous software development. For both of these methods, the so called stories are split for small tasks for iteration and the objective is to develop new releases of the software quickly and continuously. Both the XP and the LSD use CI for continuous building and testing of the software, customers are continuously included in the progress and duplicated code is reduced as much as possible.

One of the main differences in these two methods is that in the XP the development is done as pair programming, whereas the LSD is not strictly tied for that. Although, some type of lightweight code review is also used in LSD, it may be some other than pair programming. While both methods use collective ownerships and cycle through small iterations of releases and both are destined for quick delivery, in the LSD method the developers try to write as little code as possible and decide about the methods used themselves. In the LSD, the customer is not physically present with the programmers all the time.

The focus is moved more towards LSD from older agile methods such as XP [27]. LSD is not as strict and precise on its application as older agile methods like XP are [28]. Based on Xiaofeng et al. it is applied with a variety of different strategies [28]. This is both due to the fact that LSD is a relatively new agile software development method [27] and that it by its definition keeps the developers in charge of deciding the most applicable methods. The loose definition of LSD gives the developers the agility in development but does not give them that direct guidelines or specific regulations, either [28].

3 Application release automation

Many times in software development, the release process takes an extensive amount of time [11, 12]. In many cases, that means the developers must plan and execute the release for their spare time – either at weekends or at night – to avoid having the software offline in the hours of a lot of usage.

Application release automation (ARA) is mostly used as a term for packaging a release of an application and deploying it across environments and ultimately to production [29]. An ARA solution covers capabilities of environment management and modeling, deployment automation and release coordination [29]. This chapter contains the whole pipeline starting from what happens to a change in source code committed to VCS and ending with the release of the application to production environment.

Mary and Tom Poppendieck raised the questions "How long would it take your organization to deploy a change that involves just one single line of code? Do you do this on a repeatable, reliable basis?" in their book [30]. The time from deciding that you need to make a change to having it in production is known as the *cycle time*, and it is a vital metric for any project [12]. While cycle time is still measured in weeks or months in many organisation, and even the time to get the newly developed fix to a critical bug or a new feature to a testing environment takes days or weeks, this is clearly an indication that something is wrong with the release process [11, 12].

In addition to such a long cycle time, the process is in many cases not repeatable or even reliable and requires the developers to manually deploy the software to testing and staging environments before production, which is also clearly an issue with efficiency [12]. Even with complex software, it is possible to automate the building, deploying, testing and releasing as a controlled automatic process, reducing the cycle time to hours or even minutes [11, 12]. For this, Beck et al. defined the first principle of agile software as "our highest priority is to satisfy the customer through early and continuous delivery of valuable software" in Agile Manifesto [31]. The term *continuous delivery*, described in chapter 3.4, is initially defined from this principle [12].

3.1 Continuous integration

In software development, the practice of merging all the changes in the trunk of the version control system is called continuous integration (CI) [32]. CI was named and proposed by Grady Booch in 1991, although, such a frequent merging practice it is linked to nowadays was not included in the idea back then [32]. Martin Fowler defines CI as a software development practice where members of a team integrate their work frequently – usually each person integrates at least daily – leading to multiple integrations per day [33]. Each integration is verified by an automated build, also including tests, to detect integration errors as quickly as possible [32]. With CI, all the commits contain automated tests for the new or changed code and when a commit is merged to trunk, the software is automatically rebuilt and tested [33]. If the newly merged software either is not able to build or does not succeed all the

tests, changes are reverted or the developer fixes the issues otherwise rapidly.

A continuous integration framework provides for automated source repository change detection, that puts a chain of events in motion when a change is detected [34]. Typically, it first builds the software with changed source code, then runs all the unit tests and if all of those succeed, it deploys the software to a test environment for automated acceptance testing. If the automated acceptance tests encounter no errors, the newly built version is deployed to a public location for user acceptance testing (UAT). The deployment to UAT is usually kept as a separate step, though.

3.2 Deployment pipeline

CI, described above in chapter 3.1, is a huge step forward in productivity giving fast feedback on issues in software development but it on itself is not enough. While with CI, all changes are tested before deploying to a test environment, deployment pipeline takes the automation a step further [12]. The deployment pipeline is, at an abstract level, an automated manifestation of the process for getting developed software from the VCS into the hands of the end users in production environment [12]. Every change goes through a variety of stages on its way from VCS to being released [12].

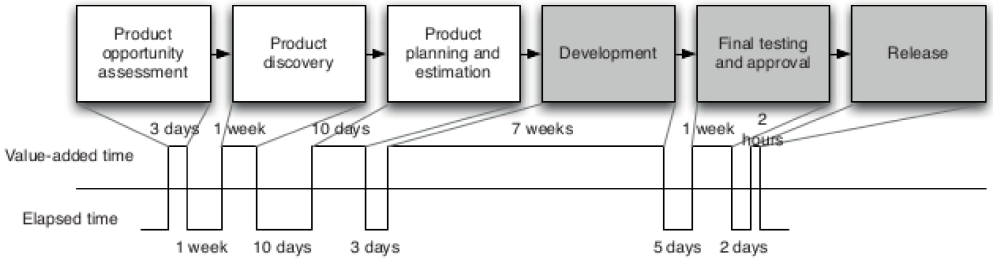


Figure 8: A value stream map of deployment pipeline process [12].

The entire process through deployment pipeline can be modelled as a value stream map [12]. A high level value stream map of deployment pipeline is shown in Figure 8. The different stages of the process and the time elapsed compared to the time some value is actually added are shown in the figure. With deployment pipeline the builds pass the value stream map multiple times on their way to the release [12]. For the deployment pipeline discussed in this thesis, the shadowed boxes on the right are the focus. In the example in the figure, the whole pipeline takes about three and a half months, in which about two and a half months are actual work. As the figure shows, there are waits between the stages of the process, possibly due to e.g. the time taken for deployment to a production-like environment.

While the value stream map shows where the most of non value-adding time is being spent and the overall process is clearly visualized, it is not very detailed or give that much information about the sub-processes in each stage. Chris Read came up with the idea to understand the deployment pipeline and the changes going through

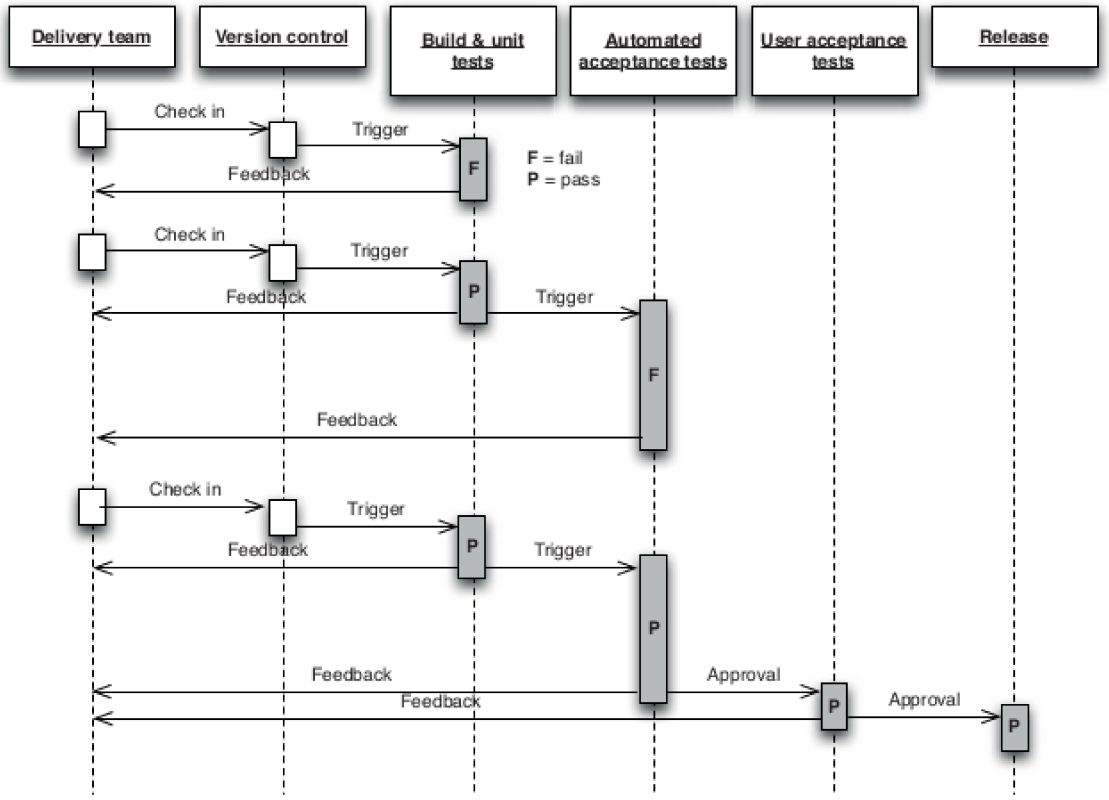


Figure 9: A sequence diagram representation of deployment pipeline [35].

with it better by visualizing the actual development parts of it as a sequence diagram [35]. Such sequence diagram is presented in Figure 9.

Development pipeline is usually divided in four stages [12]:

- **The commit stage**: An overall assertion that the software is working on a technical level. Code is compiled, unit tests are automatically run, code integrity is analyzed and if all these pass, the installer binaries (executables) are built.
- **Automated acceptance tests stages**: Automatic assertions that the software is running on a functional and non-functional level. Ensures that the software meets the behavioural requirements and needs of a user, e.g. services respond to requests correctly.
- **Manual test stages**: Assertions that the software is usable and meets the specifications and requirements of the customer. Contains both user acceptance and capacity testing. In these stages, the software is manually tested to ensure automatic tests did not miss any flaws or faulty functions.
- **Production stages**: Delivering the software to users in a staging or production environment. Usually the software is first deployed to a production-like staging

environment for the customers to test the new functionalities. When it is successfully tested, release to production.

Depending on the software, there might be some additional stages required. The trade-offs for the stages listed above are shown in Figure 10. The commit stage is basically managed with some CI tool, so CI is a vital part of the deployment pipeline.

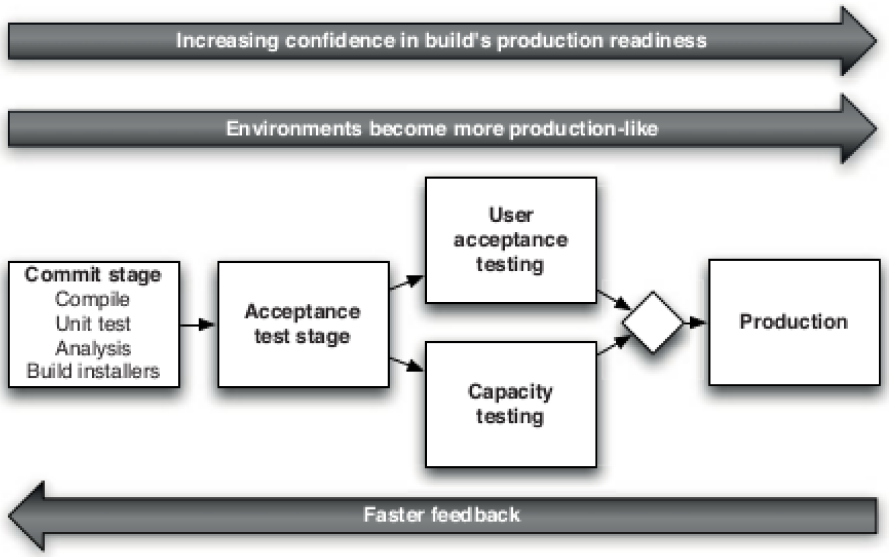


Figure 10: The trade-offs in the deployment pipeline [12].

In the deployment pipeline, the stage every change is in is visible to all members of the team [12]. Developers, testers, managers etc. can all the time check where each change is currently in the pipeline and the status if, e.g. some test or compilation has failed on some change [36]. The feedback is also gotten rapidly if something has gone wrong or a change has continued to the next stage. It is often a good practice that the changes do not continue to next stage automatically – at least not through the whole pipeline. For example, it would be good for testers to select themselves, which changes they want built in the internal testing environment before they start testing. If an iterative development method is used and many developers are resourced for the same software development project, there might be many changes waiting for testers simultaneously. It is usually easier for testers to just take one or a couple features or fixes to testing at once.

3.3 Automated testing

While unit tests run in CI to verify that each component runs successfully on its own, it must be also verified that the components work with each other as expected [12]. When many developers are developing different features for a software simultaneously, it must be well defined, how these components interact [12]. Still, while the unit tests are included in the components to make sure they are working properly on

their own, the acceptance tests must be included to ensure the interactions with other components are also working correctly. The whole flow from start to finish in the application must also be tested. This methodology is called end-to-end (E2E) testing [37].

Humble and Farley point out the importance of acceptance testing by giving an example where a malfunctioning of a system was only detected three weeks after the invalid change was made when a release candidate was deployed to test environment [12, p. 123]. In that example, a new release candidate passed all unit tests and compiled successfully on every of the 80 developers' machines. The defect was missed because the developers did not actually run the application, just compiled and run unit tests on it.

Whereas the functional testing between software components are necessary to test, software also has non-functional requirements that needs to be tested as well [12]. The capacity required by the software to install and store data and function without having too little computing power, as well as having enough network bandwidth for the peaks in usage must all be taken into account. Security vulnerabilities and service level agreements (SLAs) must be tested as well. For these non-functional tests, an additional stage is usually put on deployment pipeline.

Another important thing to ensure is that the environments are similar enough [11, 12]. At least the staging environment should be as similar as possible to production environment to ensure there is no environment specific configurations that cause issues in production [11, 12]. While there certainly are differences between environments – at least the internet protocol (IP) addresses – those differences should be in separate configuration files. That way, most of the configurations would be global but some environment specific configurations would be used based on e.g. the hostname or other environment specific variable.

3.4 Continuous delivery

Continuous delivery (CD) is the practice of automating the process for delivering the developed software from source code to production [38]. Continuous delivery is enabled through the deployment pipeline described above. Continuous integration, described in chapter 3.1, is, as a part of deployment pipeline, a part of continuous delivery. For an efficient CD, the architecturally significant requirements (ASRs) such as deployability, modifiability and testability must be met and these must be set as high priority [39].

It is sometimes required that the release to production environment is not automatically applied but needs a human to choose the time for this [11]. This might be due to customers' need to be able to schedule the release or it might even have legal or compliance rules requiring specific individual or individuals to authorize the changes to production environment. This commonly requires some authentication and authorization methods e.g. a manually inputted password, that only specific people know, to make the actual release [11]. This is where CD differs from continuous deployment. In continuous deployment, all changes that pass the automated tests are automatically deployed to production [11]. Instead, in CD, after the pipeline to

release stage is successfully completed, the change could be automatically applied to production but is necessarily not [11].

3.5 DevOps

The emphasis of collaboration and communication between software developers and other infrastructure technology (IT) professionals has increased, while processes for software delivery have become more automated and the needs for faster infrastructure changes increased [40]. The practice of such collaboration is called DevOps – a clipped compound of *development* and *operations* [40]. DevOps is originated from operations attempting to streamline and improve their effectiveness when facing increasing workloads as agile methods increased the deployment frequency and reduced the time gotten for operations [41]. While CD, the practice described in chapter 3.4, assist faster and more reliable software shipping, DevOps helps adjusting the people delivering and supporting the software [38]. Both of these practices can help optimizing, streamlining and improving working methods. A step further in automation is a concept called no operations (NoOps), that suggests an environment so automated and abstracted that there is no more need for dedicated in-house team for the infrastructure operations [41].

A practice used nowadays to keep operations' workloads from increasing too much and getting the software developers more involved in that side of deployment is called infrastructure as code (IaC) [11]. In IaC, the whole infrastructure is managed like a codebase [11]. While using IaC, all the server templates and database designs are administered through a VCS, so that all the new and old servers would have the same infrastructure and new near identical servers can be automatically setup via a pipeline like in CD [11]. This naturally requires the servers to be either virtual or in the cloud. Linux containers (LXC) are also emerging popularity with IaC, while they make it easy to package an application or a service with its dependencies to a single container without additional waste required on a full operating system [11]. They also clarify the separation between the layers of the infrastructure [11]. However, a lifespan of a LXC is not meant for as long as for a server. It is also more difficult to combine containers from separate physical hardware, while they all use the host systems kernel.

The widely used practice in IaC is that every time a new release of an application is deployed, a new server for it is also created, from the template all the old servers have been created from as well [11]. This is a way to ensure that while all the builds and test are completed, the new server environment is like the old one and the whole old server can be replaced with the new one with the newer software basically just by automatically changing the IP addresses of these two servers at the end of the pipeline. While all the environments are created from the same server templates, they are as close as identical as possible, and the new release can be done without a noticeable outage.

4 Tools for implementation

There are various different tools that help accomplish continuous delivery, or at least a part of the whole pipeline [42]. The types of tools that accomplish a part of the process cover e.g. continuous integration, application release automation, build automation and application lifecycle management (ALM) [43]. Michael Azoff listed such tools in his report [41] but due to the fact that tools have advanced enormously since 2011, that list is out of date. XebiaLabs lists a good variety of more advanced tools used nowadays on their website [44]. XebiaLabs also present the tools in so called "Periodic Table of DevOps Tools" shown in Figure 11. In the figure, different colours of elements describe, which part of the pipeline the tool can be used for. In the top right corner of an element is an acronym describing the pricing strategy of the tool. Freemium is a pricing strategy where the product or service itself is free but proprietary features and functionality are charged for [45].

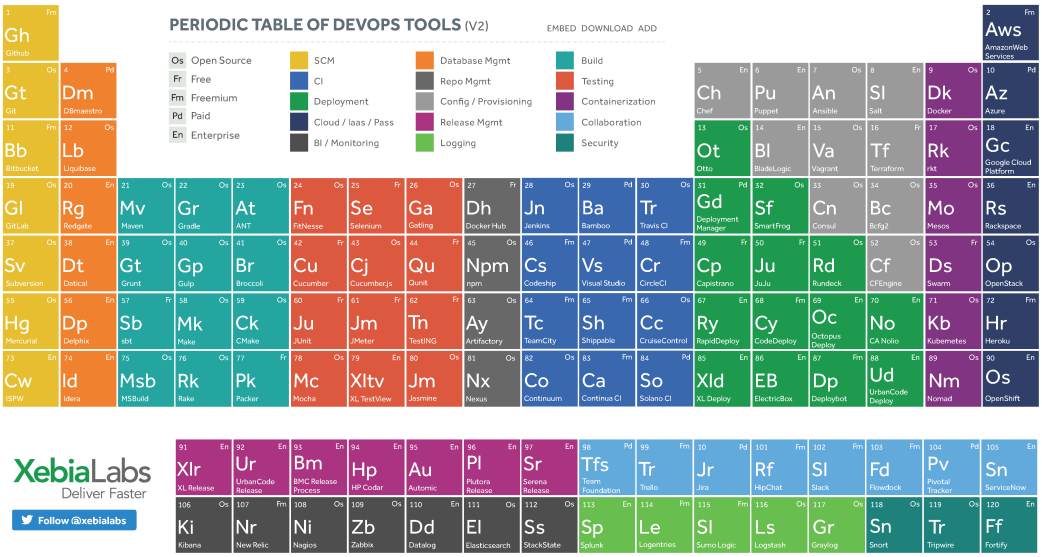


Figure 11: Periodic Table of DevOps Tools by XebiaLabs [44].

As can be seen from the figure, there is a good amount of choices for tools to accomplish different parts of the deployment pipeline – and even IaC. The build and testing tools functionalities are fulfilled as a part of CI. For the objective of this thesis, SCM (that is a synonym for version control as discussed in chapter 2.1), CI and deployment tools are used to accomplish the deployment pipeline for CD.

4.1 Version control systems

As for the VCS tools, Rob Rawson compared CVS, SVN, Git and Mercurial with each other [46]. The conclusion of that comparison is that SVN is an improved version of CVS and Git has clear speed improvements over its competitors [46]. Git is used a

bit differently than its competitors and that makes it inconvenient or difficult for some users for using it after using some of the older systems [46]. Git and Mercurial are better suited for projects containing multiple developers and more complex source tree management [46]. The using practices of Mercurial are in between SVN and Git, so it might be better suited for developers that are previously used to CVS or SVN and are not willing to adapt to Git's bit different commands. Git also provides an action called *pull request* that combines peer code review and merging a change to trunk [47]. This functionality has advantages with many CI tools but those are better discussed in chapter 5.2.

Based on a survey made by JRebel in 2012, the most responders were still using SVN as a VCS for Java projects [48]. Distributed systems were also gaining usage and developers were moving towards Git and Mercurial [48]. The distributed nature is more suitable for the distributed programming world – having multiple developers programming simultaneously and continuously [48]. New functionality has been provided and robustness and efficiency with distributed systems are increased compared to SVN and older systems [48]. The ability to check in and merge code changes while being offline are also useful features of newer distributed systems. Even given these facts, it is ultimately up to the developers, which VCS they feel the best fit for their use and all of them can be successfully used while proceeding with continuous software development.

4.2 Continuous integration tools

For the CI implementation, the tools given in Figure 11 are discussed briefly in appendix A. Each tool is – or in the case of Apache Continuum at least has been – widely used for accomplishing continuous integration. Some of these can even be used for the complete continuous delivery deployment pipeline. The tools for deployment are discussed later in chapter 4.3.

The CI tools are also shown in table 1. The platforms and natively supported builder languages and tools are shown. It is important to understand that the given builders do not list all the programming languages for the software being developed that CI tools support but only the languages and tools, the builders are – or can be – developed with. This is because of the fact that a huge variety of different programming languages can be built through e.g. shell scripts and command line and even though the CI tool would not have native support for software in given language, it is possible to develop custom builders. While the CI process with the given tools is mostly managed as workflows for different triggers, it gives a lot of possibilities to use any builders, testers etc. – either built-in, provided with some plugin or custom made – with the CI tool. Apache Continuum is left out of the comparison table while it is already retired.

Table 1: Comparison of the platforms and builders of the CI tools.

Tool	**Platform**	**Natively supported builders**
Jenkins	Web container	Ant, Cmake, Gant, Gradle, Grails, Kundo, Maven 2, MSBuild*, NAnt*, Phing, Python, Rake, Ruby, SCons, shell script, command-line
Bamboo	Web container	Ant, Bash, Grunt, Maven 1-3, MSBuild*, NAnt*, Phing, Visual Studio*, Xcode, custom script, command-line
Travis CI	Hosted	Ant, C, C++, Clojure, Elixir, Erlang, Go, Grandle, Groovy, Haskell, Java, Maven, Node.js, Perl, PHP, Python, Ruby, Rust, Scala, Smalltalk
Codeship	Hosted	Go, Java, Node.js, PHP, Python, Ruby
Visual Studio Team Services	Cross-platform	C, C++, Go, Groovy, Java, MSBuild*, Node.js, Perl, PHP, Python, Ruby, Visual Studio*
CircleCI	Hosted	Go, Java, Node.js, PHP, Python, Ruby
TeamCity	Web container	Ant, FxCop, Gradle, IntelliJ IDEA, Maven 2-3, MSBuild*, NAnt*, Rake, Visual Studio*, command-line
Shippable	Hosted	Ant, Go, Gradle, Java, Maven, Node.js, PHP, Python, Ruby, Scala
CruiseControl	Cross-platform	Ant, Java, Maven, NAnt*, Phing, Rake*, Xcode*, custom script, command-line
Continua CI	Windows	Ant, FinalBuilder*, MSBuild*, NAnt*, Powershell*, Rake*, Visual Studio*, command-line
Solano CI	Hosted, Cross-platform, Private cloud	C, C++, Clojure, Go, Java, Javascript, Node.js, PHP, Python, R, Ruby, Scala, command-line

* Only on Windows systems.

As can be seen from the brief descriptions in appendix A and comparisons in table 1, there are several different tools for accomplishing continuous integration. Different tools support different languages and work a bit differently but most of the listed tools make it possible to customize builders and that way make the continuous integration process suitable for many different occasions. The pricing strategies of the tools vary and some require a specific VCS or even specific hosting service for it. Nevertheless, the variety of different CI tools provide the necessities for continuous software development and deploying of automatically tested software for many different needs.

4.3 Deployment tools

The deployment tools from Figure 11 are listed and briefly discussed in appendix B. No tool called ElectricBox was found but ElectricFlow and ElasticBox from the ultimate list of deployment tools [44] are added instead. These are some of the most popular tools for the package deployment or full CD deployment pipeline. Some of these tools do not provide the deployment functionality themselves but allow developers to manage the pipeline with them using separate tools. Some are environment or language specific and some more generally compatible – but the latter may require more custom effort. The listed tools are also provided in table 2. Some picked restrictions, requirements and other notes are provided in the table also.

The choice of deployment or whole CD deployment pipeline tool is influenced by the project it is used for – which programming languages, environments etc. are used. Another factor in that decision is the preference of the developer team – do they like to customize a great amount of the processes and stages or should the tool contain native support for most of them. It also might be preferred to use some existing or previously used CI tool and therefore the tool used for deployment does not need to have support for the whole pipeline. In these cases, the assumption might be that the tools intended for only the deployment part of the pipeline might have better support for the part it is designed for. The tools for the whole CD might be considered having less support or features for each part of the pipeline.

Table 2: The deployment tools and notes about their restrictions and strengths.

Tool	**Notes**
Otto	Does not provide the actual functionality. Restricted to use tools by HashiCorp. Decommissioned.
Google Cloud Deployment Manager	Only for Google Cloud platform customers but free for them.
SmartFrog	Defines its own language for configurations. Only for Java-based systems.
Capistrano	Can be used for systems written in any language. Usually requires custom scripting.
JuJu	Enabled for most of the public cloud environments, OpenStack, MAAS server provisioning tool and LXC containers.
Rundeck	Does not provide tools to execute stages. External tools, command line commands or scripts must be defined.
RapidDeploy	Plugins are used for the pipeline so an external plugin must be provided for stages.
AWS CodeDeploy	Restricted to instances in Amazon environments. Can be integrated with other tools.
Octopus Deploy	Natively restricted to .NET applications but a good support for them.
CA LISA	Requires Hudson/Jenkins for building the template (the CI part or the commit stage of the pipeline).
XL Deploy	Requires external tools for executing the stages in the pipeline.
ElectricFlow	Pipeline is defined as code, so requires some amount of programming experience.
ElasticBox	Requires external tools for executing the stages in the pipeline.
Deploybot	After commit stages (CI), the stages must be defined as scripts, so scripting experience required.
UrbanCode Deploy	Requires UrbanCode Release to achieve the actual release of the application with the pipeline.

5 Implementing continuous delivery

The original plan was to implement the pipeline for an existing application for which a few new features were developed. The time frame given for the project was found too short while almost no tests at all were previously written and the structure was complex and would have needed major refactoring. It was decided that the pipeline would be implemented for a new starting project, where it would be taken into account as early as in the design phase. For the new project, there is also more time to design the pipeline successfully and start the development for the project so that development team would benefit from the pipeline starting almost from the beginning.

The project, for which the continuous delivery pipeline has been implemented, was originally developed using Visual Studio 2015 Enterprise Edition but after Visual Studio 2017 was released in March 2017, developers installed the 2017 Enterprise Edition and started developing with it within a month from the release. Visual Studio is used for building, running and debugging the software on development environments – the computers of the developers.

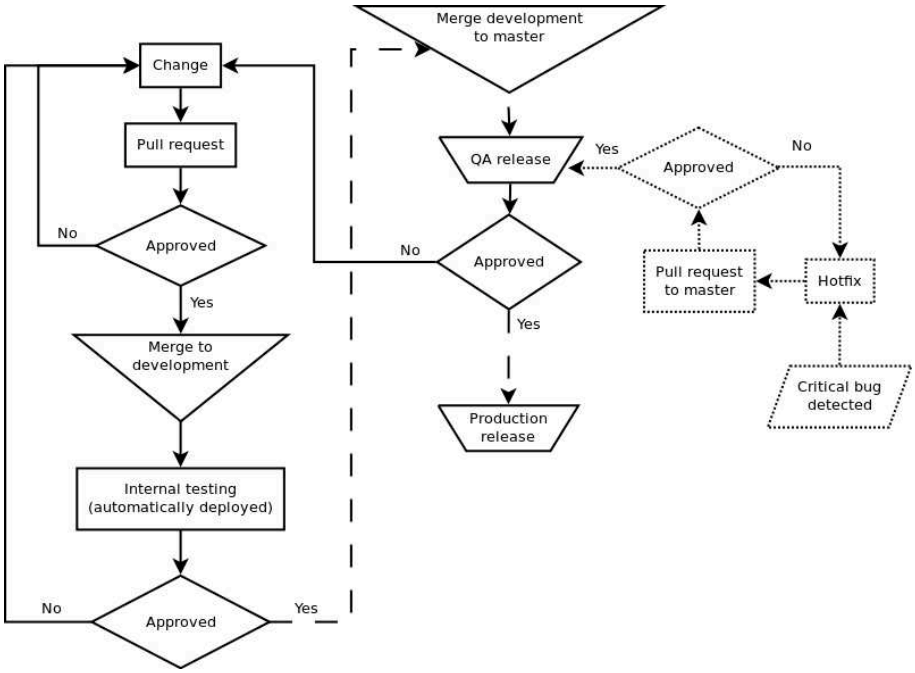


Figure 12: Process flow for a code change from been committed to a VCS branch to finally packaged and released to production environment.

The way for a change committed to VCS is shown as flowchart in Figure 12. The dashed lines express movements, that are not immediately triggered but are scheduled separately. When merging from *development* branch to *master* branch, it must be from such point where all the changes have been accepted in internal testing. The dotted lined parts on the right is a way to create so called *hotfix* – a quick fix for a critical issue that cannot wait the duration of the normal process. Even though, this is an unwanted way to abrogate the actual process, some issues might cause

the customer to lose business and the fix cannot wait its way through the normal process.

In addition to the development environments, three environments exist. Internal testing environment (DEV) is mostly for the tester to test new features and bugfixes. For DEV environment, continuous deployment is used, so every time a pull request is merged to development branch, it is directly built on CI and deployed to the internal testing environment for testing. The rest environments are quality assurance (QA) environment for QA and UAT and production environment as the actual environment for the end users. All of these three environments are run on Windows Server 2012 R2. Production environment has two servers running the application and a load balancer (LB) is routing the traffic to the server with less usage at the time. QA and production environments are built and deployed from *master* branch.

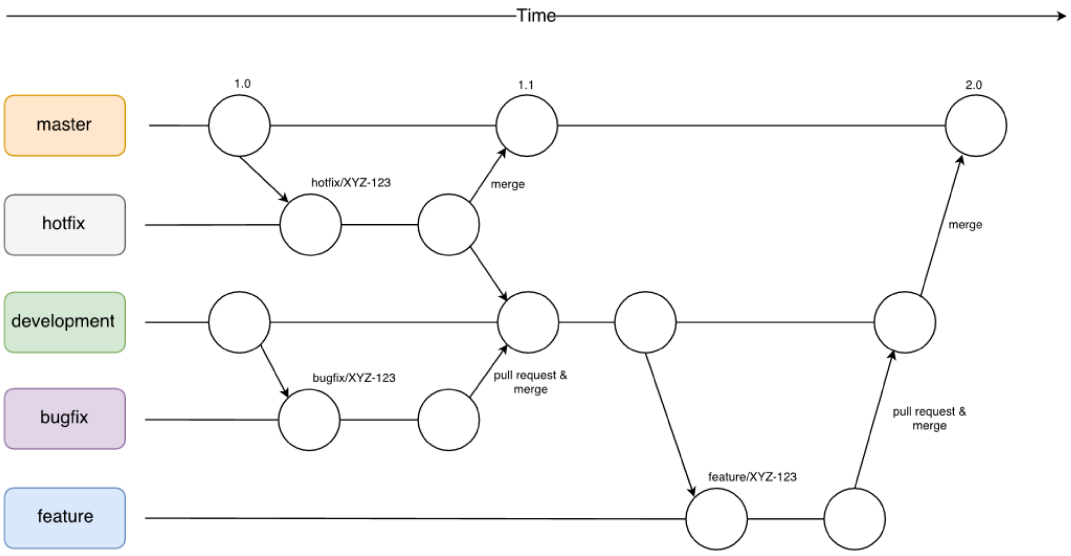


Figure 13: Branching used within the development process.

As for the VCS, an existing Mercurial instance was originally used but all the VCS history was migrated to GitHub enterprise edition that has some more advanced features used together with a CI. Two separate trunks are used. The *development* branch is used for internal testing environment and all commits are immediately built and deployed to the internal testing environment. The *master* branch is used for QA and production environments and the builds and deployments are launched manually. As shown in Figure 13, branches for features and bugfixes are created from *development* branch but hotfixes – fixes for critical bugs detected – are created directly from *master* branch to get the fixes to production as soon as possible. Even hotfixes are first deployed to QA environment for UAT, though. After the hotfix pull request is accepted and merged to *master* branch, it is also merged to *development* branch, as is shown in Figure 13. This is for the fix to stay in the application even

after the next time the *development* branch is merged to the *master* branch. It would also be possible to merge changes directly from bugfix and feature branches to master but while these branches are created from development branch, it would merge everything merged to development branch before creating the new branch from it.

The application itself is a meeting room reservation application that can be used for reserving meeting rooms and ordering catering and other services to them as well as to existing customers' rented offices. Third party company is implementing integration database with BizTalk (BT), containing application programming interfaces (APIs) developed. These APIs are used with extensible markup language (XML).

Facade and presentation layers of the whole application are managed using the pipeline implemented for this thesis. A facade pattern is a software design pattern where more complicated interfaces are simplified and wrapped in a single layer, mostly using object-oriented programming (OOP), to make it easier to use and test with the facade layers simpler methods [49]. Simultaneously the need for outside code dependencies is reduced [49]. The facade layer manages the communication between the presentation layer – the web user interface (UI) – and the BT XML APIs.

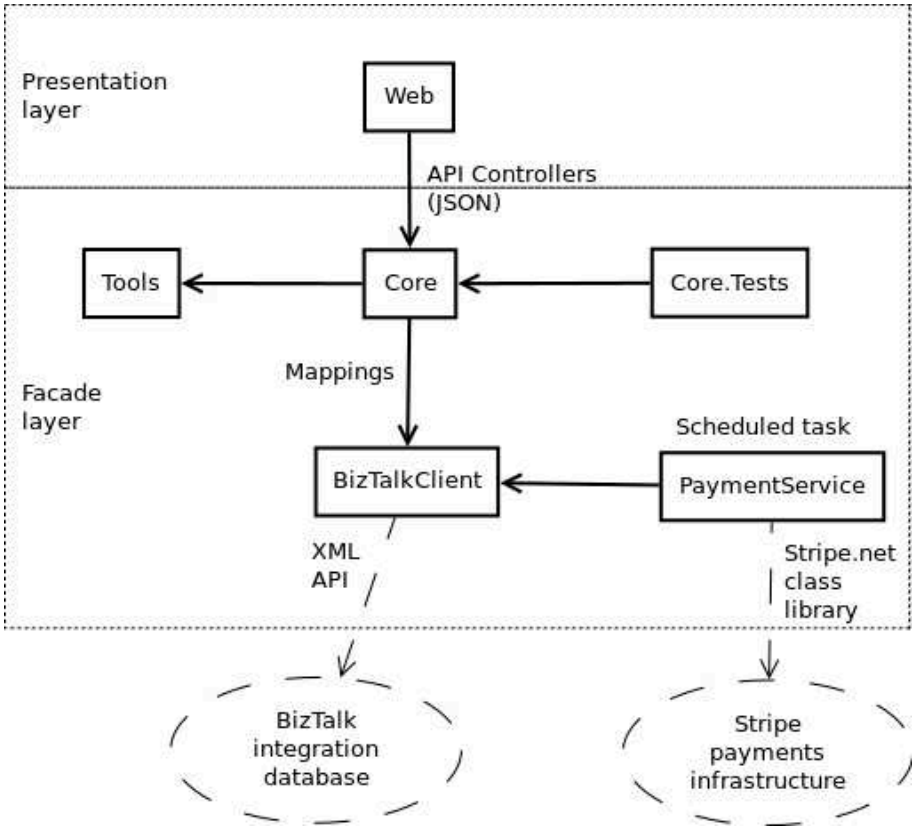


Figure 14: Architecture of the web application implemented.

The application contents are split to presentation and facade layers, as shown in Figure 14. Presentation layer – also known as front-end – contains the user interface (UI) elements and the program logic for the client – the browser in the case of web

applications. Facade layer contains the program functionality run on server side, that is not directly visible for the end users. Back-end – i.e. the server side parts of the application – are created with .NET framework by Microsoft. For the coverage of this thesis, the BT APIs, that the facade layer connects to, payment infrastructure and the rest of the system outside of the facade and presentation layers are not discussed further.

The whole web application solution is packaged in BizTalkClient, Core, Tools and Web projects. The architectural layers and components of the system are shown in Figure 14. BizTalkClient provides support for communicating with the BT APIs. Core has the facade functionality for the web application. Tools contain constants, enumerations and helper functionalities for the application. Web project has the presentation layer contents, functionality and static resources, e.g. graphical contents and translations. Web project also contains the API controllers and the unit tests for the JS functionalities. Unit tests for Core and Tools projects are included in an external Core.Tests project. All the models used in the system are defined in separate Web.Models project. The models are written with C# and similar models for TS are automatically generated from those. There is also a PaymentService project included, that provides support for payments using Stripe payments infrastructure as a separate console application. While this thesis covers the release automation of web applications only, PaymentService deployment is not covered.

As the HTML part of the software, HTML5 standard [50], created by the Web Hypertext Application Technology Working Group (WHATWG) and Word Wide Web Consortium (W3C) [51], is used. The latest evolution of CSS, CSS3 is used for the stylesheets. CSS is compiled from dynamic style sheets written in LESS. For javascript (JS) functionality, angular.js framework is used. The JS parts are written in typescript (TS), a typed superset of JS developed by Microsoft. Typescript is automatically compiled to javascript every time a change in TS file is saved. Only the TS files are included in the VCS and the JS files are recompiled during the commit stage of CD pipeline. Third party libraries for e.g. angular.js framework are included as JS files, though. Both back-end and front-end are implemented as projects in the same Visual Studio solution.

5.1 Continuous software development practice

As the practice for continuous software development, LSD is chosen, while it gives the developers more flexibility for development. While the team of people doing the development are from different backgrounds – some mostly front-end and some mostly back-end developers – the flexibility is really an important part of getting the work done fluently and getting everyone more towards fullstack knowledge, where they might develop on any part of the software – either front-end or back-end. Also, while the team for this case is quite small, containing only a technical product owner, a technical architect, a few software developers and one tester, the practice in XP, that all of the code is pair programmed, is quite difficult to achieve – even though, as discussed in chapter 2.2, it increases the code quality – it is not suitable for such a small team. Some more complex features are designed and developed together using

pair or group programming, though.

The practice that anyone could continue on any part of code is a bit out of scope here. While the developers come from different backgrounds and none has the capability to successfully develop fullstack – both back-end and front-end – on their own, the development is done split into front-end and back-end development, while the fullstack knowledge and skills are tried to be accomplished. Even though the attempt, there is no rational expectation for the developers to just adapt as fullstack developers in a small amount of time during the project development. The adaptation and learning to fluent fullstack developers might take a long time to accomplish.

While the lean practices are not exactly precise, the team organises meetings, retrospectives, every other week discussing how the development has been done and how could the practices be improved. In addition, everyday work is discussed daily, so that everyone has an idea, what each member is going to do that day and what was done the day before. The team sits in the same room so general discussion about the development occurs all the time.

Clean code practices are used in the development. With clean code, the source code functions are kept simple, consisting only of one abstraction level for the functionality [52]. Each function does only one thing and it is well described in the function name [52]. This way the source code is simpler to review by other developers. Unnecessary code comments are avoided, file sizes kept small and file structure kept clear so the source code is also easier to read and understand [52]. Using these practices simultaneously with testing the code and using a VCS, the system is kept easy to maintain and it is more effortless for new developers to get themselves familiar with the code. TSLint static analysis tool is also used for avoiding typescript readability, maintainability and functionality errors. With TSLint it is assured that the TS code stays both functioning correctly as well clean, variable and function naming is clean and separate functions and files are short enough.

5.2 Continuous integration

While there is some existing experience in using TeamCity as a CI tool in the development team and it has been proven effective in past projects, it is chosen for continuous integration for this project as well. It is also very fitting while it has no specific requirements about other tools (e.g. VCS) and it natively supports software programmed in .NET, so there is no need to provide external plugins for it.

For this project, commercial TeamCity enterprise version 9.1.5 is used. The enterprise license provides the ability to setup unlimited build configurations and it has three build agents, that can simultaneously build different projects. TeamCity is run on a Windows server and it is used from a web interface.

The instance of VCS as well as the project and branch are defined in TeamCity. It is also possible to define whether or not all the sub branches created starting from the chosen branch are also included to the CI process. The changes are checked within an interval. The duration for this interval is set in the configuration. For this project, the interval is set to check code changes every two minutes. The credentials

for authentication to VCS must also be set while the VCS project is not a public project accessible by anyone.

The build process can be started by a variety of triggers. It can be either scheduled, so that the build is run repeatedly between a separately set interval. With this option the build is run whether or not new changes occur. It can also be run every time a change is detected or there can be separate branch or branches, for which the changes that triggers the build are committed. If no triggers are configured, the build process must be started manually, by a press of a button.

After the VCS connection is successfully configured, the actual build process – or CI process – has been configured using build configuration. The build process is accomplished with build steps that are all run when a build process starts. For each step, a condition for running the step can be defined. The options for conditions are:

1. If all previous steps finished successfully.
2. Only if build status is successful.
3. Even if some of the previous steps failed.
4. Always, even if build stop command was issued.

For this project, all the steps are defined to use the first condition. The build steps are defined to use one of the predefined commands, which are either specific runners or can run a custom command line script or command.

For this project, a VCS change detecting trigger is used. While GitHub enterprise is used, when a pull request is created, it creates an additional head to VCS, preattempting a merge to the branch it is going to be merged and informs users on GitHub web UI whether merge can be automatically completed or if some merge conflicts appear. A build is also triggered on the CI by every pull request, so that it ensures the build is complete and tests succeed with each change before they are merged to trunk. If new changes are pushed either to an existing pull request or the branch it is going to be merged to, the checks and a new CI build are run again.

The build configuration steps used originally for the project in this thesis were:

1. **NuGet Restore**: Ensure that packages included in the solution are installed. To keep the VCS clean, packages folder is excluded from it, so all the packages will be downloaded and unpacked in this step.
2. **Build Web application & Compile TypeScript**: Build the Web project from the solution using MSBuild. This will also compile the typescript files from Web project to javascript for the browser to be able to run them. Core project is set as a build dependency for the Web project, so it'll be built using MSBuild too. While the Core project has Tools and BizTalkClient projects as dependencies, all of those will be built in this step too. These dependencies are shown in Figure 14.
3. **Build Payment service**: Build PaymentService project of the solution. This is done separately, while PaymentService is an external console application run

by a scheduled task. Core project dependencies used for handling payments, as well as the dependencies for those Core project functionalities are build in the console application as well.

4. **Build Core Tests**: Build the Core.Tests project from the solution. That is not set as a dependency for other projects while it is only used for running unit tests on TeamCity to ensure operability and is not an actual part of the application functionality to be published.

5. **Unit tests & coverage analysis**: Run unit tests from Core.Tests project built in the previous step and produce coverage analysis from them using predefined "xUnit.net + dotCover" meta-runner type.

6. **Chutzpah tests**: Run javascript unit tests for front-end using Chutzpah test runner. JS unit tests are written using Jasmine testing framework. These tests are run from the compiled JS files to ensure the final files used are working as expected. Code coverage report of JS files is also generated.

7. **Send Web package to SFTP server**: Send the built web application package to a seperate SFTP server that is accessible from the application servers using SSH file transfer protocol (SFTP).

8. **Send PaymentService package to SFTP server**: Send the built PaymentService console application package to the application server using SFTP.

9. **SonarQube analysis**: Run code quality analysis using SonarCube runner. It analyzes e.g. the coding standards, amount of duplicate code, unit test and code coverage, code complexity, potential bugs, commenting, design and architecture. The rules for SonarQube are defined in an external server running it. Code coverage reports from steps 5 and 6 are used to define total coverage.

NuGet package manager is used for both back-end packages, e.g. Microsoft.CSharp, and front-end packages, e.g. angular.js framework. The first step in the build configuration is meant to install all these dependencies from the repository. It does not provide the content files elsewhere than in the packages directory for the packages, so the necessary third party front-end scripts have to be included to the VCS separately. Fortunately, the scripts and stylesheets for these components do not require much space while they are basically just text files. Also, the original NuGet package installation installs most of the JS files as both original and minified version and some external files, so only the minified and actually used files are included to the solution in VCS and included in the application with ASP.NET bundle configuration.

Different publish profiles and configurations are defined for each environment separately and the ones used are selected in the second step with command line parameters. This means that similar build configuration must be defined separately for each environment, separating only with the configuration parameters that define which configuration transformation is used. For pull request validation, an additional

build configuration is used. That one is mostly like the one used for internal testing environment but does not contain steps 7 and 8, while a new version of the application should not be deployed on a pull request validation build. After the pull request is validated and reviewed, it will be merged and another build configuration triggered.

After the package is sent to the separate SFTP server in the 7th and 8th step, the application server is remotely connected and a set of powershell scripts are run to actually release the new version of the application on the server. First of these scripts determines whether a new release can be found on the SFTP server, and if so, downloads it to the application server and continues running the rest of the powershell scripts. The SonarQube analysis step is left last, while it takes some amount of time for it to go through all the files. The package should be a working version even without that step but it provides deeper analysis about the package quality and possible issues. This analysis can be used for further development and optimization. The analysis provides test coverages, clean code practice analysis results and known vulnerabilities about the code.

5.3 Package deployment

The package deployment is originally created by sending the packages to an external server via SFTP and downloaded from there to the actual application servers – as can be seen on build configuration steps 7 and 8 on chapter 5.2. New packages were then fetched to environments with custom powershell scripts run on the servers. The scripts check the SFTP server for a new package and if such exists, backs up the old version and deploys the new package on the web server. On internal testing server, the powershell scripts are run on by a scheduled task every five minutes, so the new build successfully sent from TeamCity to the SFTP server will be deployed withing five minutes from the moment, the build process succeeds. The final powershell script runs automated functionality tests with UseTrace artificial software tester. UseTrace is better explained in chapter 5.4.

With an external package deployment tool, the CI process described in chapter 5.2 will be changed, and steps 7 and 8 for SFTP transfer are replaced with a single one that runs the deployment with that tool. With a deployment tool, there is no need to manually connect to the server and run scripts after the build steps are finished but the release and other steps run by the scripts are done with the deployment tool. In the deployment tool the publish profiles are all defined and there is no need for separate build configurations for each environment. While in this application the internal testing builds from a separate VCS branch than QA and production and the package is built based on those, it is necessary to keep separate build configurations for internal testing environment. While pull request validation build configuration also have a separate VCS root configured, listening to pull request merge head branches, it is also kept as separate build configuration.

For the case of the program being written with .NET framework, and trying to keep the effort in custom scripting to a minimum, Octopus Deploy has been chosen as a deployment tool. As a .NET specific tool, there is a good support for software written with that and no custom scripting is required. Octopus Deploy has

many useful plugins to use for the project. Also, TeamCity provides a plugin to use Octopus Deploy for a build step in TeamCity.

In TeamCity, the step for Octopus Deploy deployment packages the build application as NuGet package that is then published to Octopus Deploy server. A publish API key as well as the API endpoint are configured for the communication between TeamCity and Octopus Deploy. The version number is configured to be the same as TeamCity build number so it is clearer to identify builds and packages – as well as their logs – between the systems. For the TeamCity step to know, which projects should be packaged to the built NuGet package, an external package called OctoPack must be included in those projects in the application implemented. Even though, two separate applications are built from this solution, they are both included in the same NuGet package and there is no need to specify separate deploy steps for web and console applications in TeamCity. The step for deploying the package to Octopus Deploy server checks out the projects for which OctoPack is included and packages them all to the same NuGet package. The projects set as dependencies for these projects are also included automatically. Separate OctoPack containing projects are not separated until on the Octopus Deploy server. With Octopus Deploy in use, the CI build configuration in TeamCity is changed to:

1. **NuGet Restore**
2. **Build Web application & Compile TypeScript** (including OctoPack)
3. **Build Payment service** (including OctoPack)
4. **Build Core Tests**
5. **Unit tests & coverage analysis**
6. **Chutzpah tests**
7. **Octopus Deploy**: Build OctoPacks as a single NuGet package and send it to Octopus Deply server for deployment.
8. **SonarQube analysis**

where only the 7th step is new, replacing the 7th and 8th step from build configuration in chapter 5.2 – as well as removing the need for custom powershell scripts on the servers – and the remaining steps are the same as in chapter 5.2.

5.3.1 Octopus Deploy

With Octopus Deploy, Windows services called OctopusDeploy Tentacles are installed on each environment. These services handle the new deployment installation as well as monitor the overall health of the environment, e.g. assures the operating system (OS) is up to date. The installed OctopusDeploy Tentacle service and the Octopus Deploy server are configured to use the same port and a thumbprint of a certificate by OctoDeploy Tentacle that is required to gain connection. While

production environment contains of two servers, two separate ports are used for the servers. The LB is also configured so that the deployments for the specified ports end up to the configured server so the servers are separately configured to Octopus Deploy. It is also configured that production environment deployment does not trigger for both servers in parallel simultaneously but one by one, ensuring the application is available during the deployment. The deployments for production servers are triggered manually per server so that it can be ensured, the first production server is up and running successfully before the deployment for second server is started. Octopus Deploy always communicates using secure sockets layer (SSL) encryption.

Octopus Deploy can be configured to use either listening or polling mode. In the listening mode, Octopus Deploy server deploys new package by sending it to the server where tentacle client installs it immediately. In the polling mode, the tentacle clients poll the Octopus Deploy server for new version within an interval. If new version is deployed from the web UI for given environment, it is requested from Octopus Deploy by the Tentacle. For this system, listening mode is used while it is preferred and uses less resources from the application servers.

Each environment, for which the application is to be deployed, are configured in Octopus Deploy web UI. The default deployment lifecycle between environments is set to contain all environments, starting from internal testing, then QA environment and finally production environment. While a separate build configuration is used for the internal testing environment, it is configured to use separate packages than the rest of the environments. A package intended for internal testing environment cannot be deployed to the rest and the other way around. The version numbering is also made a bit different so that it is easier to differentiate the packages meant specifically for DEV environment.

The process configuration for Octopus Deploy deployment is quite simple to understand and set up. For each step, it is selected, whether the process step is environment specific or if it is used for all environments for the same project. Also the run conditions are specified similarly as in the CI with the options:

1. Success: only run when previous steps are successful.
2. Failure: only run when previous step failed.
3. Variable: only run when the variable expression is true.
4. Always run.

The steps can be specified to run in order, after the previous step is complete or in parallel. It is also configured whether each step is run on the Octopus Deploy server or at the deployment target using OctoDeploy Tentacle.

For this solution, the CI builds separate web application and payment service console application but packages them in the same NuGet package as explained in the beginning of chapter 5.3. Even though. they are deployed in the same package, while they have separate OctoPacks, the deployment processes on Octopus Deploy must be separately configured. Still, while they are separately configured and run through a slightly different processes, both processes can be started simultaneously.

Here it is configured that every time a web application deployment is started – either automatically or manually – the payment service console application deployment starts too. This thesis concentrates on the web application deployment, so the console application will not be further discussed.

5.3.2 Web application deployment

While the web application is run as Windows Server Internet Information Service (IIS) site, the first step is a predefined IIS deployment step. For this step, it is configured, which project is to be deployed and what type of IIS deployment it is. As explained in chapter 5.2, the Web project from the solution is built for the web application, so that is also the project from the NuGet package to be deployed. Different IIS deployment types are:

1. IIS Web Site
2. IIS Virtual Directory
3. IIS Web Application

. While in this case, the whole web site is wanted to update, the deployment type 1 is selected. The web site name, application pool name, application identity, IIS bindings and authentication method that are normally configured in IIS are configured at this step. The configuration transformation is set to transform based on the defined configuration with transformation name that is set specifically for each environment. This step is run on Octopus Deploy server, which means that it starts the deployment to the deployment target.

The second step is to clean up configuration transformations. While the correct one is already transformed on the previous step as a configuration file, none of the transformations are needed anymore. This step is run after the package is deployed so it is defined that the clean is run in the installation directory. This step is run on deployment target.

The third step for deployment process is to restart the IIS application pool to clear contents from cache to prevent issues between versions, create new sessions for end users and start the new version of the application. Naturally, this step is also run on deployment target.

The fourth step is to simply send an email about new release. It is configured so, it sends the email only to the deployment team when deploying to internal testing or QA environments but also to customers' product owner on new production releases.

The last step for the deployment is to trigger automated functionality tests. This is handled by a custom powershell script step. The script sends a request to the testing system telling it to start testing the deployed application's UI. For production environment, the automated testing step is triggered only for the latter server to be deployed, while it connects to the environment with the domain name, so it is unknown, to which server the LB directs the connection to. Automated testing is better explained in chapter 5.4.

5.4 Automated functionality tests

UseTrace is chosen for the automated testing system for the application. It is an artificial software tester created especially for web applications. UseTrace is created with Selenium – a portable software testing framework for web applications. The tests are created manually by creating steps in which e.g. element is accessed, text input field is filled or user is navigated to another page. The steps are created using a virtual browser so the tests are created by actually using the application. UseTrace automatically generates selectors for the elements that are accessed but they can also be manually defined. Randomized inputs and momentary email accounts are supported so e.g. registration form can be successfully tested, including an activation email sent by the system.

UseTrace tests can be run with multiple separate virtual browsers and the reports from the test can be integrated to many group chat platforms as well as sent via email. For this thesis' project, UseTrace tests tagged as 'regression' are ran with virtual Google Chrome, Mozilla Firefox and Internet Explorer version 11 each time a new version is released to any of the environments. All tests are also ran once daily via UseTrace scheduling. The tests can also be manually triggered from UseTrace UI – either individually or all tests with a certain tag. After all tests are run, a report about how many tests succeeded is sent from UseTrace to development team via email as well as gotten to teams CA Flowdock inbox. Flowdock is used for daily communication between team members as well as the customer and third parties.

The script triggering UseTrace tests is shown in appendix C. The variables with prefix '$UseTrace' are defined in Octopus Deploy environment specific so that UseTrace knows towards which environment to test or what tests to run. Same tag cannot be used towards internal testing and production environments while in production environment, creating a new meeting generates an actual charge to a credit card.

The script sleeps for 10 seconds before starting, to ensure the site is up when testing starts. A request to execute all tests is then sent to UseTrace API to trigger configured tests. The deployment tool does not wait for all the test to execute but just confirms that a batch id for the run is gotten. That verifies that UseTrace has gotten the request and the tests will be run. The results would also be available through UseTrace API after the run has completed but that was seen as unnecessary duration to wait for the deployment tool while the report is also gotten otherwise. If the deployment would be configured to wait for the tests to complete, it would keep the process busy and no other deployment could be started before the tests were complete. The tests to be currently run with UseTrace in each environment are described in appendix D.

A common failure with internal testing releases for UseTrace was the timeout for UseTrace test steps. The BT APIs for the internal testing environment are slower and on some actions multiple requests are done in serial. Fortunately, the timeout is configurable value in UseTrace so it was increased and the tests do not fail for that cause anymore. Another common failure is due to the positioning of the UI elements. For example, a spinner element is shown over the page while asynchronous

requests are sent or user is redirected to another page. Sometimes the spinner with a transparent background colour is still visible when a notification is shown to the user, and UseTrace is unable to detect the notification element under the spinner layer. As these problems are known, the UseTrace reports with failures are investigated carefully and some actions for reducing these failures have been successfully executed. Still, these problems occur every now and then so the reports are not completely reliable.

6 Testing the implemented pipeline

While it took some time to get the firewalls configured so that TeamCity and Octopus Deploy would be able to be used for CD, the original production release, as well as a few application releases after that, were done using the previous configuration with TeamCity's SFTP step and custom powershell scripts on the servers. Finally, when the connections between Octopus Deploy server and the Tentacles were opened, it took a while before more development was done for the application.

For some new features, three two-week software development sprints were then requested by the client. The features were split to simple tasks and the work load for each of them were then estimated by the development team. Communicating with the third party managing the BT interfaces, the features were scheduled to these sprints so that the APIs would be ready for the web application development team to start the development. After each sprint, a new QA release is done and from a week to a week and a half for the clients' QA and UAT, as well as fixes to their discoveries is spent before starting the next sprint. As for this thesis, the first of these sprint is reviewed, while the others have not been finished yet, as of the time of writing.

6.1 Developing and internally testing new release

For the first new development sprint, a few new more challenging features were included, as well as some minor improvements. Each developer took one of the features to develop at the time. After the developers had finished the development and their own testing, a pull request to development branch was created. A pull request was then reviewed by one or more other team members, possible enhancements were suggested and after the pull request was approved, it was merged to development branch, after which it was instantaneously deployed to DEV environment for the tester. If the tester found issues with the feature, new issue tasks were created and they were then fixed by the first developer who finished the previous task and the newly found issue was next in line.

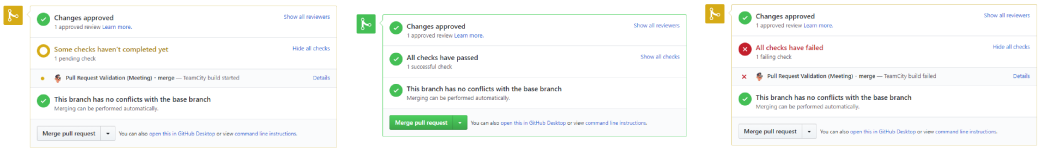
As for the development sprint, after some amount of time was passed, it was realized that the major features were more complex that originally estimated and the work load estimates were too low. While there was no willingness to lengthen the schedule, the working days for the sprint grew longer as the developers did their best to complete all the features in time. Also, the tester was able to find some combinations of actions that caused issues while the developers had not even thought about such combination. This, of course, raised the issue that the original workload estimates were not made thoroughly enough.

TeamCity as a CI tool gave good feedback about each build triggered, so that fixes to unsuccessful builds or failing tests were easy to fix. A notification of an unsuccessful build was given through an email as well as the development teams Flowdock inbox. An example on an unsuccessful build as well as the next build to fix these issues is provided in Figure 15 presenting the change log view of TeamCity. This issue was caused by a merge of two separate features simultaneously, adding them both to the same build, which caused one of the new unit tests added with one

Figure 15: An example of an unsuccessful build fixed after TeamCity notice.

of the features to fail due to the changes in the other one. Even though the fix to this issue was reviewed by other team members with another pull request, it took only a couple of hours to fix, and there was a new feature added with the new fixed build, too. While the pull request merge commits are commits merging one branch to another, TeamCity was unable to show affected files on these, so the commits on change log view showed zero files on these, as can be seen in the figure. Even so, the descriptive commit messages gave good knowledge, which commits were included and it was easy to check the actual contents of these commits from the VCS.

As where the TeamCity gave good insight for the build operability, integrating it with GitHub raised the advantage to a higher level. With GitHub integrated to TeamCity builds, all the pull requests were tested withing the pull request merge heads and it was tested that the code changes were still working if another feature was merged to the trunk. The views shown in Figure 16 visualize the view on GitHub pull request while it was built on TeamCity. After the build was triggered, the pull request showed it as in Figure 16a. Depending on the built results, the pull request was either denied or enabled. As for the Figure 16 images, the user for the GitHub is an admin, so it would be possible to merge the pull request, even though, the merged build would probably not build successfully, either. On the Figure 16b it is clearly visualized with the green color that the build was successful and the pull request can be modified. Even though, the view is from administration's point of view, the failed build on Figure 16c is seemingly not as desirable to be merged, even though it is possible.



(a) Build validation started. (b) Build validation success. (c) Build validation failed.

Figure 16: Separate build phases and status views on GitHub enterprice while integrated into the build processes.

At the end part of the development sprint the schedule for QA release was forced to be moved forward while there still existed issues found internally. After all of the internally found issues were fixed, and all the 18 UseTrace tests towards DEV environment succeeded, a pull request to merge development branch to master branch

was made and it was checked that it contained no merge conflicts. It was also assured that the new features were all included, before the actual merge was executed. After the merge was complete, it was built by TeamCity from the master branch and deployed by Octopus Deploy to the QA environment.

6.2 Quality assurance and user acceptance testing

Even though the delay, the next sprint was decided to be started as planned. When the application was finally released to QA environment, more issues than expected were still found by the customer. While the next sprint started during the previous sprints UAT, these issues were fixed as hotfixes, creating the pull requests directly toward master branch and merging them also to development branch after the pull request was approved, as discussed at the beginning of chapter 5. This left the development branch available for the next sprints contents.

While the new sprint started, and the previous sprints issues raised by the customer were also still fixed, one of the developers was assigned to handle all the issues with the previous QA release while the rest of the team were working on the new sprint contents. Fortunately, the latter sprint had fever major features included and the team was able to keep around the schedule even though the decrease in the development teams members for it. Also, the scheduling with the third party was better planned for the latter sprint, so the development was more fluent than in the previous sprint.

A production release date was decided when only few issues still existed. This, also was forced to be moved a day forward while there was no time for the customers' product owner to test the latest fixes for yet another issue found. Finally, after all the issues and discoveries by the client were fixed, a production release was approved by the client. While both QA and production environments are deployed from the build created from the master branch, the exactly same release that was lastly deployed to QA would be ready for production. For the latest QA release – the release to be deployed to production environment – all the 18 UseTrace tests succeeded.

6.3 SonarQube analysis

As the SonarQube analysis is triggered from TeamCity build configuration and QA and production environments use the same build configuration, the production release is analyzed while the package is built for the QA environment. The analysis step is also triggered each time a new version is built for the DEV environment but while that builds packages so much more frequently, each analysis report is not that thoroughly investigated.

The security rating given by SonarQube for the QA release was class A, which means there were no known vulnerabilities in the application. The reliability rating was also class A, which means that the known bug count has not increased from the previous version. The analysis showed 38 bugs in the solution, but a great amount of those were due to the compilation of typescript into javascript that caused some minor nonstandard code structure in the javascript that was analyzed. The test

coverage measure in the release was calculated at 32.9% but the development team was unable to get the SonarQube analysis step to include JS test coverage to the report. While a great amount of the application source code is written in TS and compiled to JS, the lack of JS coverage greatly reduces the overall coverage measure. The calculated technical debt of the release was three days, which also gives a class A rating, being less than 5%.

For the issue with JS test coverage, it is discussed that the front-end contents of the application would be moved from using NuGet package manager to using node.js package manager (NPM). Using NPM would also have the advantage that the necessary third party scripts would not need to be included in VCS but they would be installed and included to the solution as a CI step. That way the analysis would not have to them analyzed as a whole but only the contents necessary for the application would be included. The bundling of TS contents would also be moved from ASP.NET bundle configuration to using webpack module bundler. The test runner would be changed from chutzpah to karma test runner, from which the coverage reports are more easy to export from. A plugin called SonarTS also exist for SonarQube for analyzing typescript directly. With that plugin the bug count could be significantly reduced.

6.4 Production release

The release to production environment was scheduled to be done after 6pm at Friday evening for less usage at that time period. As the new version of the third-party BT interfaces were required to be installed first, it required for one of the web application development team to stay put until the BT APIs were successfully installed and tested. This took about ten to fifteen minutes in which time the web application was unresponsive.

After the BT APIs were installed and tested, the deployment of the web application was started. The release was first deployed to the first production server and some quick tests were manually applied there by connecting to that server via remote desktop protocol (RDP) before continuing the deployment to the second server. As all seem to work as expected, the deployment was quickly continued to the second server, after which the UseTrace tests for production environment also started. For the production environment, all the 8 UseTrace tests also succeeded.

While the BT upgrade took a relatively long time, it was decided that a maintenance break site would be put up for the following sprints' deployments to the production environment servers. It was discussed, whether the maintenance break site should be included in the deployment tool steps, so that it could be done within the rest of the deployment process using the same tool. It was not yet decided to be implemented in the process, though, and the production environment servers must currently be manually connected to for setting the maintenance break site up.

Using the maintenance break site during a deployment is not exactly agile practice. Even so, while new features would change the behaviour of the site and would not work with the previous version of the APIs, and the BT APIs are also using two separate instances under an LB, so it cannot be known to which instance the LB

will direct the traffic to. That said, while the changes in the interfaces might affect the functionality of the site, it cannot be accessed from the old version of the web application and vice versa. Nonetheless, by configuring the maintenance break site for the deployment tool, the need for connecting to the servers manually is retired for the following deployments.

It is also discussed that the LB balancing the traffic between BT instances could be configured during the release so that all the traffic would be directed to single instance for the installation period. This way, the first instance to be installed would be taken out of use for as long as the third party have installed and tested it successfully. After the first instance is installed, the LB would be configured to direct all the traffic to it, as the web application is simultaneously deployed. This way, when the web application would be released, the BT API would also be up to date and after the web application deployment, the other BT instance could be installed and tested while being isolated from the web application. After the latter instance would also be installed with the new release, the LB configuration could be reverted back to its normal behavior, directing the traffic to the instance with less existing traffic. With this process, the need for setting up a maintenance break site would disappear while the web application deployment takes the site down only for a about ten seconds during which the LB would direct the traffic to the other server. Although, the discussed changes would make the production release more fluent, it is not yet implemented, while it would require quite amount of reconfiguration on the LB.

7 Conclusions

7.1 Summary

The objective for this thesis was to choose and take in to use a continuous software development practice as well as tools for web application release automation. A literature survey for different methods and tools was performed for finding out suitable practices and tools. The evolution in software development practices from traditional waterfall model to more agile methods was described in thesis chapter 2. With iterative continuous software development methods the changes are kept smaller but they can be implemented and released more promptly. The chapter also discussed about some methods used during software development nowadays and described more closely the practices extreme programming, lean software development and adaptive software development. A comparison of these practices was also provided as well as the trend of their use lately.

The general overview and stages for application release automation were provided in chapter 3. The flow for a code change from its development for it to be released in the production environment was provided. A long cycle time – a term representing the time of this flow – has been a major issue with development and application release efficiency and application release automation is a way to reduce it greatly. The chapter provided the stages of the deployment pipeline including continuous integration, automated testing and continuous delivery. The pipeline provides the abilities to detect issues with new changes early as well as getting new features and fixes to them released quickly. In the chapter, DevOps – the usage of automated processes of software development for infrastructure as well – is also discussed briefly.

Tools for implementing application release automation within software development were presented in chapter 4. Some of the tools provide a capability for multiple separate stages of the pipeline, whereas others are more focused on a single stage. The variety of tools currently available was also acknowledged. Some of the tools were found to not actually provide the capability of any stage of the pipeline but to compose an abstraction layer for the pipeline to be implemented with other tools chosen. Tools for the deployment pipeline stages were also compared.

For the actual implementation of application release automation, a continuous software development practice as well as tools for the deployment pipeline had to be chosen. These choices and the implementation are discussed in chapter 5. Lean software development was chosen as a practice while it was found the most fitting for the resources of the development team. As a version control system, GitHub enterprise edition was chosen for the advantages it gives used together with continuous integration. As continuous integration tool, TeamCity was chosen while there was some existing experience in using it and it was found not to lack any necessary features. A previously used configuration of TeamCity without an external deployment tool was also provided in the chapter. While the software implemented was written with .NET framework, Octopus Deploy was decided to be used for deployments, as it has been originally developed especially for .NET applications. The configuration of Octopus Deploy as well as changes in TeamCity configurations were also provided in

chapter 5. As for automated testing, UseTrace was selected, while it is designed for testing web applications, which was the scope of this thesis.

After the pipeline was successfully implemented, it had to be tested in hands-on use as well. The actual software development using lean software development practice and the usage of release automation in a real-life situation during the development is reported in chapter 6. Even though, the estimations of work time were too low for the two-week development sprint, given the complexity of the features developed, the lean practice was found fit for catching up with the time later. The deployment pipeline was also found operational and suitable for the purpose.

By using a continuous software development method, the development was more agile and rapid. Using the methods, the achieved solution was more reliable and manageable than it would be by just developing it from a single plan. By using the deployment pipeline, the need for manual connections through via RDP also reduced and the deployment manners became more rapid. By using automated testing with all the stages of the pipeline, the faulty functionality with the application could be fixed at the early stage of developing, without having to find them from the production environment by the end users.

7.2 Discussion

The adaptation to lean software development was surprisingly fast. With no previous user experience, the change in the ways of working were accustomed promptly and found practical. While the features were split to simpler tasks, each of the developers could get another task to work on after the previous one was finished.

By using a peer code review on all code changes, the flaws were found early on and patched before getting any further. Even though, there were some disagreements on the structure of the code, the principles were quickly decided by the team for how to carry on. If some inoperative functionality got through the peer code review, it was at least found by the internal testing stage or from the CI build failure or, at the latest, with the automated functionality tests from the internal testing environment.

The continuous deployment on the internal testing environment was found successful for the tester to find the incorrect operations. It was found most helpful to find the incorrect functions in the development phase instead of having them found after releasing the version of the code to user acceptance testing – or worse, to the production environment. Even though, the application was – in the end – tested manually in the production environment by connecting to the first server it was deployed to via an RDP and using a browser on it, the application release automation implemented worked as desired. While there was no previous knowledge about using such pipeline, the release was surprisingly straightforward, pressing a single button on TeamCity user interface and just a couple of buttons a couple of times in the Octopus Deploy user interface.

As for the goal for the continuous deployment to be set for minimal custom programming, it was pretty well established by using the predefined runners and steps on TeamCity and Octopus Deploy. The automated testing step with UseTrace in Octopus Deploy was run with custom powershell script, it was the only one forced

to be custom written. Even though, it was a custom written script, the sample code was provided in UseTrace documentation section and only the necessary parameters for the tags, domain and browsers were required to be modified.

As for the SonarQube analysis, the results were not quite genuine. While the typescript code coverage was not accounted for in the complete code coverage analysis, the overall coverage cannot be proven appropriate. Due to the decision of using the compiled javascript code instead of the typescript code written in the coverage, the analysis results were not as actual as the code written. Using the javascript for the coverage analysis, it was found difficult for the team to get the coverage analysis included in the report, and got not yet included.

There was also just a minority of the code tested due to the difficulties raised by the Chutzpah test runner. It was found that Chutzpah demanded the references to be precisely set and even so, to have the mock data for the test written in a very specific manner. These issues caused some of the unit tests written to not compile correctly and they were forced to be left out. The usage of Chutzpah as a test runner for front-end tests might not have been the best possible idea.

7.3 Further development

While the current configuration forces the third-party scripts on the front-end to be included into the version control system, it is discussed, whether it would be better to use NPM as front-end package manager, instead. With NPM, only a package.json configuration file would be required for the VCS, containing the names and versions of the third party packages needed. An additional CI step would be added for downloading these packages from NPM within the build process.

By changing to NPM, the other front-end tests could also be run by another, more suitable test runner, such as karma. By a couple of test runs with karma, the unit tests are found to be more effortless to execute than with Chutzpah. The karma test runner is also able to run the tests directly from the written typescript files instead of the javascript files compiled. The karma test runner step could be configured to be run with Node.js. By using NPM and karma, the overall front-end source code would be required to be partially rewritten.

The configuration for the front-end code to be compiled would also be restricted from using the ASP.NET bundle configuration to a separate tool, such as webpack module bundler. Even though, the step for installing third party dependencies from NPM installs all the packages totally, webpack would only include the necessary parts of these scripts required by the code written in the application in the final packaged application.

While the typescript source code would be tested, it could be analysed, as well, using the SonarTS plugin in the SonarQube. By using the plugin, the analysis would not take in to account the compilation standard failures caused by something else than the developers. Even though, there might still be some issues with the compiled javascript code, the functionality is tested by automated tests and the analysis should not be effected by the compiler.

It is also discussed, whether the CI build configuration would be transformed as a

C# make (Cake) build. Cake build is a build automation system with a C# domain-specific language (DSL) [53]. A Cake build script would be created, including the build steps currently configured in TeamCity, as C# code [53]. While the Cake build script would be included as source code in the solution and in the VCS, there would be no need for making multiple separate configurations in the CI. The conditions could be programmed in the Cake script. By using Cake build, the CI tool could also be changed if necessary, while it supports a large variety of CI tools [53]. The configuration could also be better maintainable when included in the VCS.

For the release automation, the BT interface update causing an outage in the use of the web application is clearly an issue. The usage of temporary maintenance break site by setting it up manually via RDP is not exactly agile. The deployment pipeline will be modified so that a few additional steps for handling the maintenance break site will be prepended to the process. These steps will be used only in production environment. Predefined step templates for starting and stopping specific IIS sites, as well as for the waiting period, are found for accomplishing these.

The first new two steps – which will be the first steps for production release – will be to stop the application site and to start the maintenance break site on the IIS server. The third step after these will be a manual intervention step, i.e. an approval for the deployment process to continue after the new BT APIs are installed, tested and the third party has confirmed it is acceptable to continue the web application deployment with operational interfaces. After the approval, new steps will be added for stopping the maintenance break site and starting the web application site with previous release before the process continues the deployment of the new release immediately after the web application is running. For the deployment solution with these additional steps to be successful, the deployment will be required to be reconfigured so that the deployment for both production servers can be run in parallel. While the whole production environment must be using the maintenance break site simultaneously, the deployment will run on both until the interruption before the new interfaces are installed. After the BT APIs are ready, the interrupted deployment process will continue on both servers.

The reconfiguration of the load balancer on BT discussed at the end of chapter 6.4 would be the best solution for keeping the application release automation at its best. By directing the usage traffic between different instances of the BT interface, the application would not be offline at all. While the first instance would be installed, the usage would be directed to the other instance and after that would be done, it would be directed to the newer version before they would both be updated.

While the usage of application release automation gains a great benefit on application deployment velocity and reliability, there are still possibilities for further progression. By using DevOps and IaC practices, the whole process would become even more agile. If those practices would be used both in BT instances as well as the web application, the load balancers would not be required to be reconfigured. While the web application in the current deployment pipeline successfully releases the new version with seemingly no delay – while the deployment causes just a bit longer page reload – the usage of LXC like containers would also dispose the need for scheduled system update breaks and other non-software related issues.

While the application is implemented with .NET framework that can only be run on Windows environments, the usage of LXCs is not possible. One solution would be to migrate the solution to using .NET Core framework that is cross-platform suitable [54]. Microsoft and Docker also announced a commercial partnership in 2016 and the Docker engine is available for Windows Server 2016 [55], so by updating the environments to using Windows Server 2016, it would be also possible to use containers with the currently implemented .NET framework application.

References

[1] J. Highsmith and A. Cockburn, "Agile Software Development: The Business of Innovation," *Computer*, vol. 34, no. 9, pp. 120–122, 2011.

[2] W. Cunningham, "The WyCash Management Portfolio System," *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1993.

[3] W. A. Whitaker, "Ada - the project: the DoD high order language working group," in *ACM SIGPLAN Notices*, vol. 28, pp. 299–331, ACM, 1993.

[4] K. Beck, "Embracing Change with Extreme Programming," *Computer*, vol. 32, pp. 70–77, October 1999.

[5] B. Boehm, "Get Ready for Agile Methods, with Care," *Computer*, vol. 35, no. 1, pp. 64–69, 2002.

[6] A. Cockburn, "Agile Software Development Joins the 'Would-Be Crowd'," *Cutter IT Journal*, vol. 15, no. 1, pp. 6–12, 2002.

[7] J. Highsmith and A. Cockburn, "Agile Software Development: The People Factor," *Computer*, vol. 34, pp. 131–133, November 2001.

[8] L. Constantine, "Methodological Agility," *Software Development*, pp. 67–69, June 2001.

[9] E. Sink, "Source Control HOWTO," 2004. Available at: http://ericsink.com/scm/source_control.html (Accessed: 23.07.2016).

[10] D. Spinellis, "Version Control System," *IEEE Software*, vol. 22, no. 5, pp. 108–109, 2005.

[11] K. Morris, *Infrastructure as Code: Managing Servers in the Cloud*. O'Reilly Publishing, Inc., 2016.

[12] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Pearson Education, Inc, 2010. ISBN: 978-0-321-60191-9.

[13] A. Kolawa and D. Huizinga, *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Press, 2007. ISBN: 0-470-04212-5.

[14] L. Williams, R. Kessler, W. Cunningham, and R. Jeffries, "Strengthening the Case for Pair Programming," *IEEE Software*, July/August 2000.

[15] D. Hayes, "The Psychology Underlying the Power of Rubber Duck Debugging." Press Up Inc, 2014. Available at: https://pressupinc.com/blog/2014/06/psychology-underlying-power-rubber-duck-debugging/ (Accessed: 22.07.2016).

[16] B. George and L. Williams, "A structured experiment with test-driven development," *Information and software Technology*, vol. 46, no. 5, pp. 337–342, 2004. DOI: 10.1016/j.infsof.2003.09.011.

[17] C. Larman and V. Basili, "A history of iterative and incremental development," *IEEE Computer*, vol. 36, no. 6, pp. 47–56, 2003.

[18] R. C. Martin, "Professionalism and test-driven development," *IEEE Software*, vol. 24, no. 3, 2007.

[19] K. Beck, *Extreme Programming explained: Embrace Change*. Addison-Wesley, 2000. ISBN: 201-61641-6.

[20] M. Poppendieck and T. Poppendieck, *Lean Software Development: An Agile Toolkit*. Addison Wesley, 2003. ISBN: 0-321-15078-3.

[21] F. Ballé and M. Ballé, "Lean Development," *Business Strategy Review*, vol. 16, no. 3, pp. 17–22, 2005.

[22] S. W. Ambler and M. Lines, *Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise*. IBM Press, 2012. ISBN: 978-0-13-281013-5.

[23] J. Highsmith, "Messy, Exciting, and Anxiety-Ridden: Adaptive Software Development," *American Programmer*, no. 1, pp. 23–29, 1997.

[24] B. W. Boehm, "A Spiral Model of Software Development and Enhancement," *Computer*, vol. 21, no. 5, pp. 61–72, 1988.

[25] S. Chan, "Complex Adaptive Systems," esd.83 research seminar in engineering systems, MIT, 2001.

[26] Merriam-Webster, *Merriam-Webster's Collegiate® Dictionary*. Merriam-Webster, Inc., 11th ed., July 2003. ISBN: 978-0-87779-809-5.

[27] T. Dingsøyr, T. Dybå, and P. Abrahamsson, "A Preliminary Roadmap for Empirical Research on Agile Software Development," in *AGILE'08 Conference*, pp. 83–94, IEEE, 2008. DOI: 10.1109/Agile.2008.50.

[28] W. Xiaofeng, K. Conboy, and O. Cawley, ""Leagile" software development: An experience report analysis of the application of lean approaches in agile software development," *The Journal of Systems and Software*, vol. 85, no. 6, pp. 1287–1299, 2012.

[29] Gartner, "Application Release Automation (ARA)." Available at: http://www.gartner.com/it-glossary/application-release-automation-ara/ (Accessed: 12.08.2016).

[30] M. Poppendieck and T. Poppendieck, *Implementing lean software development: From concept to cash*. Pearson Education, 2007.

[31] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "Manifesto for Agile Software Development." Available at: http://agilemanifesto.org/ (Accessed: 05.08.2016).

[32] G. Booch, *Object Oriented Design: With Applications*. Benjamin/Cummings Publishing, 1991.

[33] M. Fowler and M. Foemmel, "Continuous Integration," *Though-Works*, 2006. Available at: https://www.thoughtworks.com/continuous-integration (Accessed: 25.07.2016).

[34] S. Stolberg, "Enabling Agile Testing with Continuous Integration," in *Agile Conference, 2009. AGILE'09*, pp. 369–374, IEEE, 2009.

[35] C. Read, "Continuous Integration, Build Pipelines and Continuous Deployment." Talk given at DevOpsDays, October 2009.

[36] P. M. Duvall, "Continuous Delivery: Patterns and Antipatterns in Software Lifecycle," *DZone Refcardz*, 2012. Available at: http://www.dccia.ua.es/dccia/inf/asignaturas/MADS/lecturas/10_Continuous_Delivery_Dzone_Refcardz.pdf (Accessed: 05.08.2016).

[37] Technopedia™, "Techopedia - The IT Education Site." Available at: https://www.techopedia.com/ (Accessed: 26.08.2016).

[38] P. Swartout, *Continuous Delivery and DevOps - A Quickstart Guide*. Packt Publishing Ltd., 2014. ISBN 978-1-78439-931-3.

[39] C. Lianping, "Towards Architecting for Continuous Delivery," in *12th Conference on Software Architecture (WICSA 2015)*, (Montréal, Canada), IEEE, 2015.

[40] F. Erich, C. Amrit, and M. Daneva, "Cooperation between Information System Development and Operations," *International Conference on Product-Focused Software Process Improvement*, pp. 277–280, 2014.

[41] M. Azoff, "DevOps: Advances in Release Management and Automation," tech. rep., Ovum, 2011.

[42] A. Phillips, "The Continuous Delivery Pipeline — What it is and Why it's so Important in Developing Software," July 2014. Available at: http://devops.com/2014/07/29/continuous-delivery-pipeline/ (Accessed: 12.08.2016).

[43] A. Binstock, "Continuous Delivery: The Agile Successor," September 2014. Available at: http://www.drdobbs.com/architecture-and-design/continuous-delivery-the-agile-successor/240169037 (Accessed: 12.08.2016).

[44] XebiaLabs, "The Ultimate List of Deployment Tools." Available at: https://xebialabs.com/the-ultimate-devops-tool-chest/deployment/ (Accessed: 16.08.2016).

[45] J. L. M. de la Iglesia and J. E. L. Gayo, *Web 2.0: The Business Model*, ch. 6. Doing business by selling free services, pp. 89–95. Springer, 2009. ISBN-13: 978-0-387-85894-4.

[46] R. Rawson, "Comparison of Version Control Software: SVN, Git, Mercurial." Available at: https://biz30.timedoctor.com/git-mecurial-and-cvs-comparison-of-svn-software/ (Accessed: 19.08.2016).

[47] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social coding in GitHub: transparency and collaboration in an open software repository," in *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pp. 1277–1286, ACM, 2012.

[48] A. Koblentz, "DevProd Report Revisited: Version Control Systems in 2013," February 2013. Available at: https://zeroturnaround.com/rebellabs/devprod-report-revisited-version-control-systems-in-2013/ (Accessed: 20.08.2016).

[49] E. Freeman, E. Robson, B. Bates, and K. Sierra, *Head first design patterns*. O'Reilly Media, Inc., 2004. ISBN: 978-0-596-00712-6.

[50] I. Hickson, R. Berjon, S. Faulkner, T. Leithead, E. D. Navada, E. OCónnor, and S. Pfeiffer, "HTML5," October 2014. Available at: https://www.w3.org/TR/2014/REC-html5-20141028/ (Accessed: 02.09.2016).

[51] W3C, "Frequently Asked Questions (FAQ) about the future of XHTML," July 2009. Available at: https://www.w3.org/2009/06/xhtml-faq.html (Accessed: 02.09.2016).

[52] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education, 2009.

[53] .NET Foundation and contributors, "Official Cake Website." Available at: https://cakebuild.net/ (Accessed: 15.10.2017).

[54] C. Simmons, ".NET Core Framework - Go Cross-Platform with the .NET Framework," 2015. Available at: https://msdn.microsoft.com/en-us/magazine/dn913184.aspx (Accessed: 15.10.2017).

[55] S. Johnston, "Docker Announces Commercial Partnership with Microsoft to Double Container Market by Extending Docker Engine to Windows Server," 2016. Available at: https://blog.docker.com/2016/09/docker-microsoft-partnership/ (Accessed: 15.10.2017).

[56] Jenkins, "Official Jenkins Website." Available at: https://jenkins.io/ (Accessed: 20.08.2016).

[57] Atlassian, "Official Bamboo Website." Available at: https://www.atlassian.com/software/bamboo/ (Accessed: 20.08.2016).

[58] Travis CI community, "Official Travis CI Website." Available at: https://travis-ci.org/ (Accessed: 20.08.2016).

[59] Codeship Inc, "Official Codeship Website." Available at: https://codeship.com/ (Accessed: 20.08.2016).

[60] Microsoft, "Official Microsoft Visual Studio Continuous Integration Website." Available at: https://www.visualstudio.com/en-us/features/continuous-integration-vs.aspx (Accessed: 20.08.2016).

[61] CircleCI, "Official CircleCI Website." Available at: https://circleci.com/ (Accessed: 20.08.2016).

[62] JetBrains, "Official TeamCity Website." Available on: https://www.jetbrains.com/teamcity/ (Accessed: 19.08.2016).

[63] Shippable Inc, "Official Shippable Website." Available at: https://circleci.com/ (Accessed: 20.08.2016).

[64] CruiseControl development team, "Official CruiseControl Website." Available at: http://cruisecontrol.sourceforge.net/ (Accessed: 20.08.2016).

[65] The Apache Software Foundation, "Official Apache Continuum Website." Available at: https://continuum.apache.org/ (Accessed: 20.08.2016).

[66] VSoft Technologies, "Official Continua CI Website." Available at: https://www.finalbuilder.com/continua-ci (Accessed: 20.08.2016).

[67] Solano Labs, "Official Solano Labs Website." Available at: https://www.solanolabs.com/ (Accessed: 20.08.2016).

[68] HashiCorp, "Official Otto Website." Available at: https://www.ottoproject.io/ (Accessed: 26.08.2016).

[69] Google, "Official Google Cloud Deployment Manager Website." Available at: https://cloud.google.com/deployment-manager/ (Accessed: 26.08.2016).

[70] SmartFrog, "Official SmartFrog Website." Available at: http://www.smartfrog.org/ (Accessed: 26.08.2016).

[71] Capistrano, "Official Capistrano Website." Available at: http://capistranorb.com/ (Accessed: 26.08.2016).

[72] Canonical Group Ltd, "Official JuJu Website." Available at: https://jujucharms.com/ (Accessed: 26.08.2016).

[73] Rundeck, "Official Rundeck Website." Available at: http://rundeck.org/ (Accessed: 26.08.2016).

[74] MidVision Ltd, "Official RapidDeploy Website." Available at: http://www.midvision.com/product/rapiddeploy/ (Accessed: 27.08.2016).

[75] Amazon Web Services, "Official AWS CodeDeploy Website." Available at: https://aws.amazon.com/codedeploy/ (Accessed: 26.08.2016).

[76] Octopus Deploy, "Official Octopus Deploy Website." Available at: https://octopus.com/ (Accessed: 27.08.2016).

[77] C. Technologies, "Official CA LISA® Release Automation Website." Available at: http://www.ca.com/us/products/ca-release-automation.html (Accessed: 27.08.2016).

[78] XebiaLabs, "Official XL Deploy Website." Available at: https://xebialabs.com/products/xl-deploy/ (Accessed: 26.08.2016).

[79] Electric Cloud, "Official ElectricFlow Website." Available at: http://electric-cloud.com/products/electricflow/ (Accessed: 27.08.2016).

[80] ElasticBox, "Official ElasticBox Website." Available at: https://elasticbox.com/ (Accessed: 27.08.2016).

[81] Wildbit LLC, "Official Deploybot Website." Available at: https://deploybot.com/ (Accessed: 27.08.2016).

[82] IBM, "Official UrbanCode Deploy Website." Available at: http://www-03.ibm.com/software/products/en/ucdep (Accessed: 26.08.2016).

A Continuous integration tools

Jenkins

Jenkins is a free open source continuous integration tool written in Java. The project was originally forked from Hudson in 2011 after a dispute with Oracle. Jenkins is run in a servlet container on a server and it supports all the most used SCM systems. Jenkins is originally made for Java projects but plugins exist that can extend the use of projects in different languages. Jenkins can be used as a simple CI tool or as a whole CD pipeline hub with additional plugins. [56]

Bamboo

Bamboo is a continuous integration tool made by Atlassian and written in Java. It is free for open source projects but commercial organizations are charged by the amount of build agents. Bamboo supports building in any language using any build tool. Atlassian has also made it possible to migrate from Jenkins to Bamboo using an importer. Bamboo can also be used as a complete CD pipeline. [57]

Travis CI

The company behind Travis CI was founded in 2011 and experienced significant growth in 2012. Travis CI is a distributed service used for continuous integration. It is used to build, test and deploy projects hosted at GitHub. For open source and private projects, a free testing is available at travis-ci.org. Although it is a free software and available for the users of GitHub, the company notes that casual users are unlikely able to integrate that to their own projects. Travis CI supports a good variety of languages for build. [58]

Codeship

Codeship is a continuous integration tool supporting Dart, Go, Java and JVM based languages, Node.js, PHP, Python and Ruby. Codeship can also be used as a complete CD pipeline and it has native support for docker containerization platforms. [59]

Visual Studio

Visual Studio's continuous integration is a part of Visual Studio team services, a software as a service (SaaS) offering of Visual Studio on Microsoft Azure cloud computing platform. [60]

CircleCI

CircleCI is a cloud-based continuous integration tool that supports Go, Haskell, Java, Node.js, PHP, Python, Ruby/Rails and Scala languages. With CircleCI, a GitHub project is connected to it and the status of tests with a build can be

monitored from GitHub. CircleCI can also be integrated to a Bitbucket project. Docker containerization platform is also supported. CircleCI is used for continuous integration and deployment of web and mobile applications. [61]

TeamCity

JetBrains TeamCity is a Java-based management and CI tool. It is priced with a freemium strategy. The supported programming languages are Java, .NET, Ruby and XCode. Other languages are not natively supported but many plugins exist providing the support for e.g. C++, Python, PHP and Node.js. [62]

Shippable

Shippable is a platform for the whole CD deployment pipeline. It supports most of the version control systems, including SaaS systems. With Shippable, any part of the pipeline can be orchestrated separately from the rest, having e.g. continuous integration server or automated E2E tests in customers' own database behind firewalls. [63]

CruiseControl

CruiseControl is an open source CI tool and an extensible framework for custom build processes. CruiseControl is written in Java and developed by volunteers. It contains pre-built builders: Ant, NAnt, Maven, Phing, Rake and Xcode, and a catch-all exec builder that can be used to run any command line tool or script. [64]

Continuum

Apache Continuum™ is a CI tool that is built on Apache Maven software project management and comprehension tool. Continuum is written in Java and uses build definitions from shell scripts for building the projects. It especially supports Apache Maven projects but can be configured for any other type of project too. Every time a change that fails the build is commited, Continuum sends an email to developers notifying about the need to fix the error. Apache Continuum has been retired since 18th May 2016. [65]

Continua CI

Continua CI is a CI and release management server tool for Windows. With Continua CI the process is managed as condition programmed workflows. It has support for Visual Studio, MSBuild, Ant, Nant, Rake, FinalBuilder, Git, Hg and Svn. [66]

Solano CI

Solano CI is a tool that supports seamlessly Java, C/C++, Python, Ruby, Javascript, Scala, PHP and Go languages. It works with Git and Mercurial. SolanoLabs has

different versions of CI products: Solano CI SaaS, that contains the whole CD pipeline, and Solano Private CI, a private virtual appliance for customers' own infrastructures. Solano Private CI can be run from anywhere, e.g. either from customers' own datacenter or from public cloud provider. [67]

B Deployment tools

Otto

Otto is a single solution for developing and deploying applications by providing a high level appfile which describes the tools for different stages of the deployment pipeline. Other tools by HashiCorp, including Vagrant, Packer, Terraform, Consul and Vault, are used for managing development environments, provisioning and configuring the infrastructure and packaging and deploying the application. Otto has been decommissioned since 19th September 2016 while HashiCorp felt that it had not lived up to the expectations. It is no longer supported or maintained, even though it is still available. [68]

Google Cloud Deployment Manager

Google Cloud Deployment Manager declares the requirements for the deployment process by yaml files. Python and Jinja2 templates are used for the control of the flow between stages. JavaScript object notation (JSON) schema is used for defining and constraining parameters. The configurations are treated like code, so that the deployment process is repeatable and rapid. Deployment manager is available only for Google Cloud platform customers without additional charges. [69]

SmartFrog

SmartFrog is a software framework for configuring, deploying and managing distributed software systems written in Java. SmartFrog is open source software and can be used also on commercial purposes for free. Additional components may be created but modifications on SmartFrog core or its public component library must be contributed back to open source if they are to be distributed. SmartFrog defines its own language for defining configurations, system modeling capabilities and system configurations. It has a runtime system that interprets the language and configures and deploys according to the templates. [70]

Capistrano

Capistrano is a remote server automation tool written in Ruby. It can be used to deploy software written in any language, although, it might require an extension to support special deployment requirements. Capistrano is a scriptable tool that can be used to execute arbitrary tasks on a deployment workflow. [71]

JuJu

JuJu is a modeling tool for applications and services that enables modeling, configuring, deploying and managing applications in the cloud environment. JuJu includes providers for all the major public clouds, OpenStack, MAAS server provisioning tool

and LXC containers. It can also be used for scaling the environment of the software. [72]

Rundeck

Runbook is a job scheduler and runbook automation provider for the full deployment pipeline. With Runbook, single jobs are defined for each stage of the pipeline to execute any set of commands, scripts or tools. These jobs can be triggered either by a scheduler or on-demand via either application programming interface (API) or manually from web interface. The output and process of each change in the pipeline can also be monitored from the web interface. The web interface allows easy access to run jobs and monitor processes so the job execution responsibility can be handed off to e.g. any given developer, analyser or tester. [73]

RapidDeploy

RapidDeploy is a release and deployment automation tool created by MidVision. RapidDeploy provides a plugin-centric architecture that supports integrating with a variety of different applications. The release modeling and packaging is made so, that it is possible to rollback to an older package version on any environment manually at the click of a button. Modeling also allows scaling the deployments to any environment without the need to change configurations. [74]

AWS CodeDeploy

AWS CodeDeploy is a service for code deployment provided by Amazon Web Services. Deployment can be made to any instance on Amazon. It tracks application change on changes by the configurable rules and reduces the downtime of the application during deployment. CodeDeploy can be integrated with any existing application or even another CD tool, e.g. Jenkins. [75]

Octopus Deploy

Octopus Deploy is a deployment automation server tool for .NET applications and services. It has built-in features specifically for .NET application deployment, so there is no need to manually build the deployment stage routines. With Octopus Deploy, clients called Octopus Tentacles are installed on every server that the software is going to be deployed to. Octopus Deploy is intended for the deployment part of the pipeline only, so an external built server – a CI tool – is also required. [76]

CA LISA Release Automation

CA Technologies has acquired Nolio Zero Touch Deployment™ to better their CA LISA® solutions for release automation. With CA LISA, the agile applications are released rapidly and continuously. It can be used for fully automating releases

through different environments and E2E orchestration on composite releases. Uses Hudson/Jenkins for CI. [77]

XL Deploy

XL Deploy is a deployment automation and standardizion tool made by XebiaLabs. It is used with other tools for each stage in the deployment pipeline. Its architecture is agentless, so there is no need to install proprietary agent software on each of the server environments used within the pipeline. It keeps track of all changes in the system and rolling back to earlier release or undeploying new one are possible with a simple click of a button. [78]

ElectricFlow

ElectricFlow is a DevOps release automation tool created by ElectricCloud. The processes for deployment pipeline are defined as code. Applications, pipelines, environments and releases are modeled and the changes on CD pipeline can be tracked, managed and the access to separate stages can be defined based on roles. ElectricFlow is offered as a free community edition that runs in VirtualBox as well as a commercial enterprise edition. [79]

ElasticBox

ElasticBox is a CD tool for building a customized stack of components from chosen tools and using them togeter as a CD pipeline. ElasticBox can be used in any public or private cloud environment. The simpler cloud edition is provided for free but for more secure, stable and supported enterprise solution is priced inidividually for the customers. [80]

Deploybot

Deploybot is a deployment tool assembled by Wildbit. It is used for compiling or building code and executing scripts before, after or during the deployment process. This way it is possible to manage the whole pipeline for any type of application. Deploybot also provides support for pre-defined and completely customized Docker containers. The code building and script execution before, during or after the deployment is still a beta version, at least by August 27th 2016. [81]

UrbanCode Deploy

UrbanCode Deploy is a deployment pipeline orchestration and automation tool made by IBM. It contains a graphical editor for creating automated deployment processes as well as management features that permit deployments on both configuration only solutions and code and configuration solutions. Changes are traceable and

deployment artifacts stored. UrbanCode Deploy is used with UrbanCode Release to reach the full continuous delivery functionality. [82]

C Script used to trigger UseTrace test automation

```
# Security protocol must be set for the connection to UseTrace
    ↪ to success
[Net.ServicePointManager]::SecurityProtocol = [Net.
    ↪ SecurityProtocolType]::Tls12

# Ensure site is up
Start-Sleep -s 10

# Start UseTrace
$startBatchResource = "https://api.usetrace.com/api/project/
    ↪ $UseTraceProjectId/execute_all/?key=$UseTraceApiKey"
$startBatchRequest = @"
{
    "baseUrl": "$UseTraceDomain",
    "requiredCapabilities":[
        {"browserName":"chrome"},
        {"browserName":"firefox"},
        {"browserName":"internet explorer", "version": 11}
    ],
    "tags":["$UseTraceTag"]
}
"@

$ts = Get-Date -Format HH:mm.ss
Write-Output "$ts UseTrace starting towards $UseTraceDomain with
    ↪  tests tagged '$UseTraceTag'."
$batchId = Invoke-RestMethod -ContentType "application/json" -
    ↪ Method Post -Uri $startBatchResource -Body
    ↪ $startBatchRequest -Verbose -Debug
if ($batchId) {
    Write-Output "Tests started. You can see the results at http
        ↪ ://api.usetrace.com/api/results/$batchId/xunit after
        ↪ the run has finished."
} else {
    Write-Error "Could not get batch ID for UseTrace run"
}
```

D UseTrace tests run for each environment

1. Admin creates a catering order for user's behalf and cancels it
   - DEV, QA
2. Admin reserves a meeting room on user's behalf and cancels it
   - DEV, QA
3. Anonymous user is requested to log in on the way to reservation page
   - DEV, QA, Production
4. Anonymous user views a product card
   - DEV, QA
5. Basic search returns something from a specific city
   - DEV, QA, Production
6. Existing user is able to change user information
   - DEV, QA, Production
7. Existing user is able to log in
   - DEV, QA, Production
8. External user adds or modifies credit card details
   - DEV, QA
9. The front page contains required details
   - DEV, QA
10. New user is able to register and log in
    - DEV, QA, Production
11. New user registers and adds credit card
    - DEV, QA
12. New user registers and reserves a meeting room and then cancels it
    - DEV, QA
13. User creates a catering order and cancels it
    - DEV, QA

14. User can filter search results by room name on advanced search page
    - DEV, QA, Production
15. User changes his/her password and is able to log in with new password
    - DEV, QA, Production
16. User changes language from English to Finnish and the site contents change
    - DEV, QA, Production
17. User is able to reserve a meeting room with sauna and admin confirms reservation
    - DEV, QA
18. User reserves a meeting room and cancels it late and the cancellation cost (50%) is correct
    - DEV, QA